

# FDPS Fortran Interface Tutorial

Daisuke Namekata, Masaki Iwasawa, Keigo Nitadori, Ataru Tanikawa,  
Takayuki Muranushi, Long Wang, Natsuki Hosono, and Jun-ichiro Makino

Paricle Simulator Research Team, RIKEN Center for Computational  
Science, RIKEN

## 0 Contents

---

<b>1 Change Log</b>	<b>5</b>
<b>2 Overview</b>	<b>6</b>
<b>3 Getting Started</b>	<b>7</b>
3.1 Environment . . . . .	7
3.2 Necessary software . . . . .	7
3.2.1 Standard functions . . . . .	7
3.2.1.1 Single thread . . . . .	7
3.2.1.2 Parallel processing . . . . .	7
3.2.1.2.1 OpenMP . . . . .	7
3.2.1.2.2 MPI . . . . .	8
3.2.1.2.3 MPI+OpenMP . . . . .	8
3.2.2 Extensions . . . . .	8
3.2.2.1 Particle Mesh . . . . .	8
3.3 Install . . . . .	8
3.3.1 How to get the software . . . . .	8
3.3.1.1 The latest version . . . . .	9
3.3.1.2 Previous versions . . . . .	9
3.3.2 How to install . . . . .	9
3.4 How to compile and run the sample codes . . . . .	10
3.4.1 Gravitational $N$ -body simulation . . . . .	10
3.4.1.1 Summary . . . . .	10
3.4.1.2 Move to the directory with the sample code . . . . .	10
3.4.1.3 Edit Makefile . . . . .	10
3.4.1.4 Run make . . . . .	12
3.4.1.5 Run the sample code . . . . .	13
3.4.1.6 Analysis of the result . . . . .	13
3.4.1.7 To use Phantom-GRAPE for x86 . . . . .	15

3.4.1.8	To use PIKG . . . . .	15
3.4.2	SPH simulation code . . . . .	15
3.4.2.1	Summary . . . . .	15
3.4.2.2	Move to the directory with the sample code . . . . .	15
3.4.2.3	Edit Makefile . . . . .	16
3.4.2.4	Run make . . . . .	16
3.4.2.5	Run the sample code . . . . .	16
3.4.2.6	Analysis of the result . . . . .	16
<b>4</b>	<b>How to use</b> . . . . .	<b>18</b>
4.1	<i>N</i> -body simulation code . . . . .	18
4.1.1	Location of source files and file structure . . . . .	18
4.1.2	User-defined types and user-defined functions . . . . .	18
4.1.2.1	FullParticle type . . . . .	18
4.1.2.2	calcForceEpEp . . . . .	20
4.1.2.3	calcForceEpSp . . . . .	21
4.1.3	The main body of the user program . . . . .	22
4.1.3.1	Creation of an object of type <code>fdps_controller</code> . . . . .	22
4.1.3.2	Initialization and Termination of FDPS . . . . .	22
4.1.3.3	Creation and initialization of FDPS objects . . . . .	23
4.1.3.3.1	Creation of FDPS objects . . . . .	23
4.1.3.3.2	Initialization of <code>DomainInfo</code> object . . . . .	23
4.1.3.3.3	Initialization of <code>ParticleSystem</code> object . . . . .	24
4.1.3.3.4	Initialization of <code>Tree</code> object . . . . .	24
4.1.3.4	Initialization of particle data . . . . .	24
4.1.3.5	Time integration loop . . . . .	25
4.1.3.5.1	Domain Decomposition . . . . .	25
4.1.3.5.2	Particle Exchange . . . . .	26
4.1.3.5.3	Interaction Calculation . . . . .	26
4.1.3.5.4	Time integration . . . . .	26
4.1.3.6	Update of particle data . . . . .	27
4.1.4	Log file . . . . .	28
4.2	SPH simulation code with fixed smoothing length . . . . .	28
4.2.1	Location of source files and file structure . . . . .	28
4.2.2	User-defined types and user-defined functions . . . . .	28
4.2.2.1	FullParticle type . . . . .	28
4.2.2.2	EssentialParticleI(J) type . . . . .	29
4.2.2.3	Force type . . . . .	30
4.2.2.4	calcForceEpEp . . . . .	31
4.2.3	The main body of the user program . . . . .	34
4.2.3.1	Creation of an object of type <code>fdps_controller</code> . . . . .	34
4.2.3.2	Initialization and termination of FDPS . . . . .	34
4.2.3.3	Creation and initialization of FDPS objects . . . . .	34
4.2.3.3.1	Creation of necessary FDPS objects . . . . .	35
4.2.3.3.2	Initialization of the domain information object . . . . .	35
4.2.3.3.3	Initialization of <code>ParticleSystem</code> object . . . . .	36

4.2.3.3.4	Initialization of Tree objects . . . . .	36
4.2.3.4	Time integration loop . . . . .	36
4.2.3.4.1	Domain Decomposition . . . . .	36
4.2.3.4.2	Particle Exchange . . . . .	36
4.2.3.4.3	Interaction Calculation . . . . .	36
4.2.4	Compilation of the program . . . . .	37
4.2.5	Execution . . . . .	37
4.2.6	Log and output files . . . . .	38
4.2.7	Visualization . . . . .	38
<b>5</b>	<b>Sample Codes</b>	<b>39</b>
5.1	<i>N</i> -body simulation . . . . .	39
5.2	SPH simulation with fixed smoothing length . . . . .	52
<b>6</b>	<b>Extentsions</b>	<b>69</b>
6.1	P <sup>3</sup> M code . . . . .	69
6.1.1	Location of sample code and working directory . . . . .	69
6.1.2	User-defined types . . . . .	69
6.1.2.1	FullParticle type . . . . .	69
6.1.2.2	EssentialParticleI type . . . . .	70
6.1.2.3	Force type . . . . .	71
6.1.2.4	calcForceEpEp . . . . .	71
6.1.2.5	calcForceEpSp . . . . .	73
6.1.3	Main body of the sample code . . . . .	74
6.1.3.1	Creation of an object of type <code>fdps_controller</code> . . . . .	74
6.1.3.2	Initialization and Termination of FDPS . . . . .	75
6.1.3.3	Creation and initialization of FDPS objects . . . . .	75
6.1.3.3.1	Creation of necessary FDPS objects . . . . .	75
6.1.3.3.2	Initialization of FDPS objects . . . . .	76
6.1.3.4	Generation of a distribution of particles . . . . .	77
6.1.3.4.1	Domain Decomposition . . . . .	77
6.1.3.4.2	Particle Exchange . . . . .	78
6.1.3.5	Interaction Calculation . . . . .	78
6.1.3.6	Calculation of relative energy error . . . . .	79
6.1.4	Compile . . . . .	79
6.1.5	Run . . . . .	79
6.1.6	Check the result . . . . .	79
<b>7</b>	<b>Practical Applications</b>	<b>81</b>
7.1	<i>N</i> -body/SPH code . . . . .	81
7.1.1	How to run the sample code . . . . .	81
7.1.1.1	Move to the directory the sample code is placed . . . . .	81
7.1.1.2	File structure of the sample code . . . . .	82
7.1.1.3	Edit Makefile . . . . .	82
7.1.1.4	Create particle data using MAGI . . . . .	85
7.1.1.5	Run make . . . . .	86

7.1.1.6	Run the sample code . . . . .	86
7.1.1.7	Analysis of the result . . . . .	86
7.1.2	Springel's SPH scheme . . . . .	87
7.1.3	User-defined types . . . . .	88
7.1.3.1	FullParticle type . . . . .	89
7.1.3.2	EssentialParticle type . . . . .	90
7.1.3.3	Force type . . . . .	91
7.1.4	Interaction functions . . . . .	92
7.1.4.1	Interaction function for the gravity calculation . . . . .	92
7.1.4.2	Interaction function for the density calculation . . . . .	96
7.1.4.3	Interaction function for the calculation of pressure-gradient acceleration . . . . .	100
7.1.5	Main body of the sample code . . . . .	102
7.1.5.1	Creation of an object of type <code>fdps_controller</code> . . . . .	103
7.1.5.2	Initialization and termination of FDPS . . . . .	103
7.1.5.3	Creation and initialization of FDPS objects . . . . .	104
7.1.5.3.1	Creation and initialization of <code>ParticleSystem</code> objects . . . . .	104
7.1.5.3.2	Creation and initialization of <code>DomainInfo</code> object . . . . .	104
7.1.5.3.3	Creation and initialization of <code>TreeForForce</code> objects . . . . .	104
7.1.5.4	Setting initial condition . . . . .	105
7.1.5.5	Domain decomposition . . . . .	106
7.1.5.6	Particle exchange . . . . .	106
7.1.5.7	Interaction calculations . . . . .	106
7.1.5.8	Time integration . . . . .	109
<b>8</b>	<b>User Supports</b>	<b>110</b>
8.1	Compile-time problem . . . . .	110
8.2	Run-time problem . . . . .	110
8.3	Other cases . . . . .	110
<b>9</b>	<b>License</b>	<b>111</b>

# 1 Change Log

---

- 2017/2/8
  - English version created.
- 2018/07/11
  - Typographical error correction in Section 4:
    - \* Some of included source codes are unintentionally truncated (Sec. 4.1, Sec. 4.2)
    - \* Names of some directories are wrong
- 2018/08/29
  - Description of *N*-body/SPH sample code is added. (Sec. 7.1)
- 2018/08/31
  - Description of the Phantom-GRAPE library for x86 is added. (Sec. 3.4.1.7)
- 2019/07/19
  - Description of *N*-body/SPH sample code is updated. (Sec. 7.1)
- 2020/8/16
  - Description of PIKG is added. (Sec. 3.4.1.8)
- 2020/8/28
  - Web links for the initial condition files used in *N*-body/SPH sample code are changed. (Sec. 7.1)
- 2020/09/02
  - Description for `init_tree` that leads to misunderstanding is fixed. The sample codes are also fixed.

## 2 Overview

In this section, we present the overview of Framework for Developing Particle Simulator (FDPS) and FDPS Fortran interface . FDPS is an application-development framework which helps the application programmers and researchers to develop simulation codes for particle systems. What FDPS does are calculation of the particle-particle interactions and all of the necessary works to parallelize that part on distributed-memory parallel computers with near-ideal load balancing, using hybrid parallel programming model (uses both MPI and OpenMP). Low-cost part of the simulation program, such as the integration of the orbits of particles using the calculated interaction, is taken care by the user-written part of the code.

FDPS support two- and three-dimensional Cartesian coordinates. Supported boundary conditions are open and periodic. For each coordinate, the user can select open or periodic boundary.

The user should specify the functional form of the particle-particle interaction. FDPS divides the interactions into two categories: long-range and short-range. The difference between two categories is that if the grouping of distant particles is used to speedup calculation (long-range) or not (short range).

The long-range force is further divided into two subcategories: with and without a cutoff scale. The long range force without cutoff is what is used for gravitational  $N$ -body simulations with open boundary. For periodic boundary, one would usually use TreePM, P<sup>3</sup>M, PME or other variant, for which the long-range force with cutoff can be used.

The short-range force is divided to four subcategories. By definition, the short-range force has some cutoff length. If the cutoff length is a constant which does not depend on the identity of particles, the force belongs to “constant” class. If the cutoff depends on the source or receiver of the force, it is of “scatter” or “gather” classes. Finally, if the cutoff depends on both the source and receiver in the symmetric way, its class is “symmetric”. Example of a “constant” interaction is the Lennard-Jones potential. Other interactions appear, for example, SPH calculation with adaptive kernel size.

The user writes the code for particle-particle interaction kernel and orbital integration using Fortran 2003 .

# 3 Getting Started

In this section, we describe the first steps you need to do to start using FDPS and FDPS Fortran interface . We explain the environment (the supported operating systems), the necessary software (compilers etc), and how to compile and run the sample codes.

## 3.1 Environment

FDPS works on Linux, Mac OS X, Windows (with Cygwin).

## 3.2 Necessary software

In this section, we describe software necessary to use FDPS, first for standard functions, and then for extensions.

### 3.2.1 Standard functions

we describe software necessary to use standard functions of FDPS. First for the case of single-thread execution, then for multithread, then for multi-nodes.

#### 3.2.1.1 Single thread

- make
- A C++ compiler (We have tested with gcc version 8.x and later)
- Ruby (necessary for PIKG. We have tested with version 3.0.2)
- A Fortran compiler that supports Fortran 2003 Standard and that are interoperable with the above C++ compiler (We have tested with gcc version 8).
- Python 2.7.5 or later, or, Python 3.4 or later (correct operation is not guaranteed for older Python versions)

#### 3.2.1.2 Parallel processing

##### 3.2.1.2.1 OpenMP

- make
- A C++ compiler with OpenMP support (We have tested with gcc version 8)
- A Fortran compiler with OpenMP support (it must support Fortran 2003 Standard and be interoperable with the above C++ compiler. We have tested with gcc version 8).
- Python 2.7.5 or later, or, Python 3.4 or later (correct operation is not guaranteed for older Python versions)

### 3.2.1.2.2 MPI

- make
- A C++ compiler which supports MPI version 1.3 or later. (We have tested with Open MPI 1.6.4 and K compiler version 1.2.0)
- A Fortran compiler which supports MPI version 1.3 or later (it also must support Fortran 2003 Standard and be interoperable with the above C++ compiler. We have tested with OpenMPI 1.6.4).
- Python 2.7.5 or later, or, Python 3.4 or later (correct operation is not guaranteed for older Python versions)

### 3.2.1.2.3 MPI+OpenMP

- make
- A C++ compiler which supports OpenMP and MPI version 1.3 or later. (We have tested with Open MPI 1.6.4 and K compiler version 1.2.0)
- A Fortran compiler which supports OpenMP and MPI version 1.3 or later (it also must support Fortran 2003 Standard and be interoperable with the above C++ compiler. We have tested with OpenMPI 1.6.4).
- Python 2.7.5 or later, or, Python 3.4 or later (correct operation is not guaranteed for older Python versions)

## 3.2.2 Extensions

Current extension for FDPS is the “Particle Mesh” module. We describe the necessary software for it below.

### 3.2.2.1 Particle Mesh

- make
- A C++ compiler which supports OpenMP and MPI version 1.3 or later. (We have tested with Open MPI 1.6.4)
- FFTW 3.3 or later

## 3.3 Install

In this section we describe how to get the FDPS software and how to build it.

### 3.3.1 How to get the software

We first describe how to get the latest version, and then previous versions. We recommend to use the latest version.

### 3.3.1.1 The latest version

You can use one of the following ways.

- Using browsers

1. Click “Download ZIP” in <https://github.com/FDPS/FDPS> to download FDPS-master.zip
2. Move the zip file to the directory under which you want to install FDPS and unzip the file (or place the files using some GUI).

- Using CLI (Command line interface)

- Using Subversion:

```
$ svn co --depth empty https://github.com/FDPS/FDPS
$ cd FDPS
$ svn up trunk
```

- Using Git

```
$ git clone git://github.com/FDPS/FDPS.git
```

### 3.3.1.2 Previous versions

You can get previous versions using browsers.

- Previous versions are listed in <https://github.com/FDPS/FDPS/releases>. Click the version you want to download it.
- Extract the files under the directory you want.

### 3.3.2 How to install

Because FDPS is a header library<sup>1)</sup>, you do not have to execute the `configure` command. All you need to do is to expand the archive of FDPS in some directory and to setup the include PATH when you compile your codes. An actual procedures can be found in Makefiles of the sample codes explained in § 3.4.

When using FDPS from Fortran, you first must create interface programs to FDPS based on user’s codes. Its procedure is described in Chap. 6 of the specification document `doc_spec_ftn_en.pdf`. Makefiles of the sample codes are written so that the interface programs are automatically generated when `make` are running. We recommend that users use Makefiles of the sample codes as a reference when making your own Makefile.

---

<sup>1)</sup>A library that consists of header files only.

## 3.4 How to compile and run the sample codes

We provide two samples: one for gravitational  $N$ -body simulation and the other for SPH. We first describe gravitational  $N$ -body simulation and then SPH. Sample codes do not use extensions.

### 3.4.1 Gravitational $N$ -body simulation

#### 3.4.1.1 Summary

Through the following steps one can use this sample.

- Move to the directory `$(FDPS)/sample/fortran/nbody`. Here, `$(FDPS)` denotes the highest-level directory for FDPS (Note that FDPS is not an environmental variable). The actual value of `$(FDPS)` depends on the way you acquire the software. If you used the browser, the last part is “FDPS-master”. If you used Subversion or Git, it is “trunk” or “FDPS”, respectively.
- Edit `Makefile` in the current directory `$(FDPS)/sample/fortran/nbody`.
- Run the `make` command to create the executable `nbody.out`.
- Run `nbody.out`
- Check the output.

In addition, we describe the way to use Phantom-GRAPE for x86.

#### 3.4.1.2 Move to the directory with the sample code

Move to `$(FDPS)/sample/fortran/nbody`.

#### 3.4.1.3 Edit Makefile

In the directory, there are two Makefiles: `Makefile` and `Makefile.intel`. The former is for GCC and the latter is for the Intel compilers. In this section, we mainly describe `Makefile` in detail and give an usage note on `Makefile.intel` at the end of this section.

First, we describe the default setting of `Makefile`. There are four Makefile variables that need to be set when compiling the sample code. They are the following. `FC` that stores the command to run a Fortran compiler, `CXX` that stores the command to run a C++ compiler, and `FCFLAGS` and `CXXFLAGS`, in which compiler options for both compilers are stored. The initial values of these variables are as follows:

```
FC=gfortran
CXX=g++
FCFLAGS = -std=f2003 -O3 -ffast-math -funroll-loops -finline-functions
CXXFLAGS = -O3 -ffast-math -funroll-loops $(FDPS_INC)
```

where `$(FDPS_INC)` is the variable storing the include PATH for FDPS. It is already set in this Makefile and you do not need to modify it here.

An executable file can be obtained by executing the `make` command after setting the above four Makefile variables appropriately. Edit `Makefile` according the following descriptions. The changes depend on if you use OpenMP and/or MPI.

- Without both OpenMP and MPI
  - Set the variable `FC` the command to run your Fortran compiler
  - Set the variable `CXX` the command to run your C++ compiler
- With OpenMP but not with MPI
  - Set the variable `FC` the command to run your Fortran compiler with OpenMP support
  - Set the variable `CXX` the command to run your C++ compiler with OpenMP support
  - Uncomment the line `FCFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`
  - Uncomment the line `CXXFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`
- With MPI but not with OpenMP
  - Set the variable `FC` the command to run your Fortran compiler that supports MPI
  - Set the variable `CXX` the command to run your C++ compiler that supports MPI
  - Uncomment the line `FCFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`
  - Uncomment the line `CXXFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`
- With both OpenMP and MPI
  - Set the variable `FC` the command to run your Fortran compiler that supports both OpenMP and MPI
  - Set the variable `CXX` the command to run your C++ compiler that supports both OpenMP and MPI
  - Uncomment the line `FCFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`
  - Uncomment the line `FCFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`
  - Uncomment the line `CXXFLAGS += -DPARTICLE_SIMULATOR_THREAD_PARALLEL -fopenmp`
  - Un comment the line `CXXFLAGS += -DPARTICLE_SIMULATOR_MPI_PARALLEL`

Next, we describe useful information when users use this Makefile to compile users' codes. Most important variables when using this Makefile are `FDPS_LOC`, `SRC_USER_DEFINED_TYPE`, and `SRC_USER`. The variable `FDPS_LOC` is used to store the PATH of the top directory of FDPS. Based on the value of `FDPS_LOC`, this Makefile automatically sets a lot of variables related to FDPS, such as the PATH of the directory storing FDPS source files and the PATH of the Python script to generate Fortran interface. Thus, users should set appropriately. The variable `SRC_USER_DEFINED_TYPE` is used to store a list of names of Fortran files in which

user-defined types are implemented, while the variable `SRC_USER` is used to store a list of names of Fortran files in which all the rest are implemented. The reason why we divide users' source files as above is to avoid needless recompilation of FDPS (as a result, we can reduce time required to compile and link users' codes): Because FDPS Fortran interface programs are generated based on user-defined types, we need to recompile of FDPS only when files specified by `SRC_USER_DEFINED_TYPE` are modified. However, there is one thing users should be careful of. When there are dependencies between files specified by `SRC_USER_DEFINED_TYPE` or `SRC_USER`, users must describe these dependencies in Makefile. As for the way of describing dependencies in Makefile, please see the manual of GNU make, for example.

Finally, we describe the usage note for `Makefile.intel`. Except for the initial values of Makefile variables, `Makefile.intel` has the same structure as that of `Makefile`. Hence, users can make use of `Makefile.intel` in the same way as `Makefile` by modifying the values of the variables appropriately. The followings are things to keep in mind when editing `Makefile`:

- `/opt/intel/bin` should be replaced by the PATH of a directory that stores Intel compilers in your computer system.
- `/opt/intel/include` should be replaced by the PATH of a directory that stores header files used by Intel compilers.
- By default, the value of the variable `LDFLAGS` is `-L/opt/intel/lib/intel64 -L/usr/lib64 -lifport -lifcore -limf -lsvml -lm -lipgo -lirc -lirc_s`. Among them, the option `-lifcore`<sup>2)</sup> is necessary for the Intel C++ compiler to link C++ objects and Fortran objects<sup>3)</sup>. When the Intel compiler's libraries are not in the library PATH of the system, users need to specify libraries as `-L/opt/intel/lib/intel64 -L/usr/lib64 -lifport -limf -lsvml -lm -lipgo -lirc -lirc_s`, where `/opt/intel/lib/intel64` is the PATH of directory that stores the Intel compiler's libraries, `/usr/lib64` is the PATH of directory storing the library `libm`. These PATHs depend on the systems users use and therefore users must modify these appropriately. Note that libraries required to compile users' codes (`-l*`) may change depending on the version of Intel compilers and please confirm these.
- As of writing this (2016/12/26), the compile option that invokes OpenMP support is either `-openmp` or `-qopenmp` depending the version of Intel compilers. Recent compilers use the latter option (if the former is specified in this case, the compiler issues a warning of "deprecated").
- Depending on computer systems, all of the necessary settings except for the specification of the option `-lifcore` may be done by environment variables such as `PATH`, `CPATH`, `LD_LIBRARY_PATH`.

#### 3.4.1.4 Run make

Type "make" to run `make`. In the process of `make`, Fortran interface programs are first generated and then they are compiled together with the sample codes.

---

<sup>2)</sup>`libifcore` is an Intel compiler's Fortran runtime library.

<sup>3)</sup>We have tested this with Intel compilers (ver. 17.0.0 20160721).

### 3.4.1.5 Run the sample code

- If you are not using MPI, run the following in CLI (terminal)

```
$ ./nbody.out
```

- If you are using MPI, run the following in CLI (terminal)

```
$ MPIRUN -np NPROC ./nbody.out
```

Here, `MPIRUN` should be `mpirun` or `mpiexec` depending on your MPI configuration, and `NPROC` is the number of processes you will use.

Upon normal completion, the following output log should appear in `stderr`. The exact value of the energy error may depend on the system, but it is okay if its absolute value is of the order of  $1 \times 10^{-3}$ .

```
time: 9.5000000000E+000, energy error: -3.8046534069E-003
time: 9.6250000000E+000, energy error: -3.9711750200E-003
time: 9.7500000000E+000, energy error: -3.8223429428E-003
time: 9.8750000000E+000, energy error: -3.8843099298E-003
MemoryPool::finalize() is completed!
***** FDPS has successfully finished. *****
```

### 3.4.1.6 Analysis of the result

In the directory `result`, files “`snap0000x-proc000y.dat`” have been created. These files store the distribution of particles. Here, `x` is an integer indicating time and `y` is an integer indicating MPI process number (`y` is always 0 if the program is executed without MPI). The output file format is that in each line, index of particle, mass, position (`x, y, z`) and velocity (`vx, vy, vz`) are listed.

What is simulated with the default sample is the cold collapse of an uniform sphere with radius three expressed using 1024 particles. Using `gnuplot`, you can see the particle distribution in the `xy` plane at `time=9`:

```
$ cd result
$ cat snap0009-proc* > snap0009.dat
$ gnuplot
> plot "snap0009.dat" using 3:4
```

By plotting the particle distributions at other times, you can see how the initially uniform sphere contracts and then expands again. (Figure 1).

To increase the number of particles to 10000, set the value of the parameter variable `ntot` (defined in the subroutine `f_main()` in the file `f_main.F90`) to 10000, then recompile the sample codes, and run the executable file again.

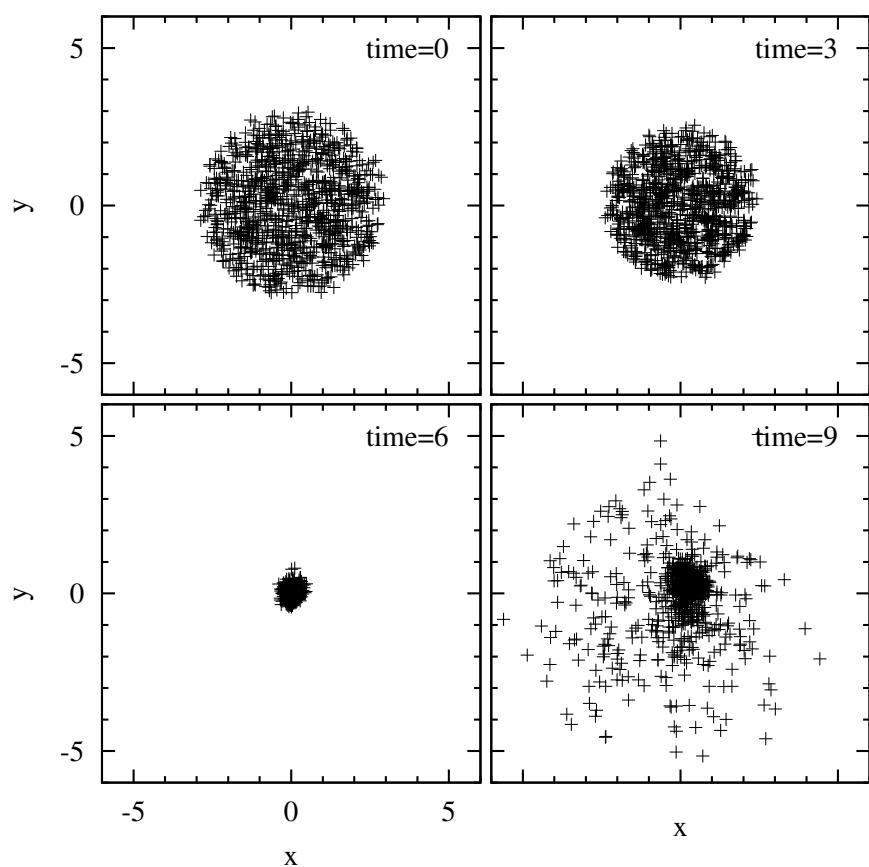


Figure 1:

### 3.4.1.7 To use Phantom-GRAPE for x86

If you are using a computer with Intel or AMD x86 CPU, you can use Phantom-GRAPE for x86.

Move to the directory `$(FDPS)/src/phantom_grape_x86/G5/newton/libpg5`, edit the Makefile there (if necessary), and run make to build the Phantom-GRAPE library `libpg5.a`.

Then go back to directory `$(FDPS)/sample/fortran/nbody`, edit Makefile and remove “`#`” at the top of the line

`"#use_phantom_grape_x86 = yes"`, and (after removing the existing executable) run make again. (Same for with and without OpenMP or MPI). You can run the executable in the same way as that for the executable without Phantom GRAPE.

The performance test on a machine with Intel Core i5-3210M CPU @2.50GHz (2 cores, 4 threads) indicates that, for N=8192, the code with Phantom GRAPE is faster than that without Phantom GRAPE by a factor a bit less than five.

### 3.4.1.8 To use PIKG

PIKG (<https://github.com/FDPS/PIKG>) is a tool to generate a highly-optimized, two-body inter-particle interaction calculation kernel for particle simulations from a simple description of the interaction using a DSL (Domain Specific Language).

In order to use kernels generated by PIKG, open Makefile in directory `$(FDPS)/sample/fortran/nbody` and remove `#` at the top of the line `#use_pikg_x86 = yes`. Then, (after removing the existing executable) run `make`. (Same for with and without OpenMP or MPI). You can run the executable in the same way as that for the executable without PIKG.

In the default, PIKG generates kernels in `reference` mode. In this mode, unoptimized kernels are generated. To generate kernels optimized for specific architectures such as AVX2 and AVX-512, change `CONVERSION_TYPE` in Makefile and remove `#` at the top of the line containing `*FLAGS` (where `*` is the usual regular expression symbol).

## 3.4.2 SPH simulation code

### 3.4.2.1 Summary

Through the following steps one can use this sample.

- Move to the directory `$(FDPS)/sample/fortran/sph`.
- Edit Makefile in the current directory (`$(FDPS)/sample/fortran/sph`).
- Run make command to create the executable `sph.out`.
- Run `sph.out`.
- Check the output.

### 3.4.2.2 Move to the directory with the sample code

Move to `$(FDPS)/sample/fortran/sph`.

### 3.4.2.3 Edit Makefile

Edit Makefile following the same description described in § 3.4.1.3.

### 3.4.2.4 Run make

Type “make” to run `make`. As in  $N$ -body sample code, in the process of `make`, Fortran interface programs are first generated. Then, they are compiled together with SPH sample codes.

### 3.4.2.5 Run the sample code

- If you are not using MPI, run the following in CLI (terminal)

```
$ ./sph.out
```

- If you are using MPI, run the following in CLI (terminal)

```
$ MPIRUN -np NPROC ./sph.out
```

Here, `MPIRUN` should be `mpirun` or `mpiexec` depending on your MPI configuration, and `NPROC` is the number of processes you will use.

Upon normal completion, the following output log should appear in `stderr`.

```
***** FDPS has successfully finished. *****
```

### 3.4.2.6 Analysis of the result

In the directory `result`, files “`snap0000x-proc0000y.dat`” have been created. These files store the distribution of particles. Here, `x` and `y` are integers that indicate time and MPI process number, respectively. When executing the program without MPI, `y` is always 0. The output file format is that in each line, index of particle, mass, position (`x`, `y`, `z`), velocity (`vx`, `vy`, `vz`), density, internal energy and pressure are listed.

What is simulated is the three-dimensional shock-tube problem. Using `gnuplot`, you can see the plot of the `x`-coordinate and density of particles at time=40:

```
$ cd result
$ cat snap0040-proc* > snap0040.dat
$ gnuplot
> plot "snap0040.dat" using 3:9
```

When the sample worked correctly, a figure similar to Figure 2 should appear.

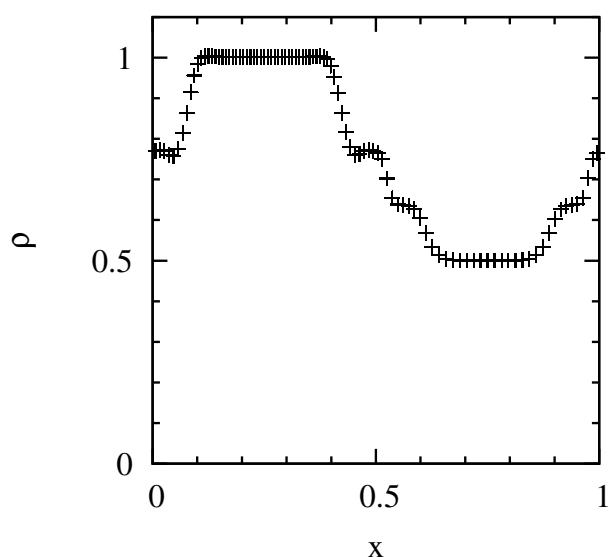


Figure 2:

## 4 How to use

In this section, we describe the sample codes used in previous section (§ 3) in more detail. Especially, the explanation will focus mainly on derived data types that users must define (hereafter, **user-defined types**) and how to use APIs of Fortran interface to FDPS. In order to avoid duplication of explanation, some matters are explained in § 4.1 only, where we explain the *N*-body sample code. Therefore, we recommend users who are interested in SPH simulation only to read § 4.1.

### 4.1 *N*-body simulation code

#### 4.1.1 Location of source files and file structure

The source files of the sample code are in the directory `$(FDPS)/sample/fortran/nbody`. The sample code consists of `user_defined.F90` where user-defined types are described, and `f_main.F90` where the other parts of *N*-body simulation code are implemented. In addition to these, there are two Makefiles: `Makefile` (for GCC) and `Makefile.intel` (for Intel compilers).

#### 4.1.2 User-defined types and user-defined functions

In this section, we describe the details of derived data types and subroutines that users must define when performing an *N*-body simulation with FDPS.

##### 4.1.2.1 FullParticle type

You must define a `FullParticle` type. `FullParticle` type should contain all physical quantities necessary for an *N*-body simulation. Listing 1 shows the implementation of `FullParticle` type in our sample code (see `user_defined.F90` ).

Listing 1: `FullParticle` type

---

```

1  type, public, bind(c) :: full_particle !$fdps FP,EPI,EPJ,Force
2      !$fdps copyFromForce full_particle (pot,pot) (acc,acc)
3      !$fdps copyFromFP full_particle (id,id) (mass,mass) (pos,pos)
4      !$fdps clear id=keep, mass=keep, pos=keep, vel=keep
5      integer(kind=c_long_long) :: id
6      real(kind=c_double) mass !$fdps charge
7      type(fdps_f64vec) :: pos !$fdps position
8      type(fdps_f64vec) :: vel !$fdps velocity
9      real(kind=c_double) :: pot
10     type(fdps_f64vec) :: acc
11 end type full_particle

```

---

When developing a simulation code with FDPS Fortran interface, users must specify which user-defined type (`FullParticle`, `EssentialParticleI`, `EssentialParticleJ`, and `Force` types) a derived data type corresponds to. In FDPS Fortran interface, this is done by adding a **FDPS directive**, which is a Fortran's comment text with a special format, to a derived data type. Because `FullParticle` type is used as `EssentialParticleI` type, `EssentialParticleJ` type,

and **Force** type in this sample code, a FDPS directive specifying that the derived data type acts as any types of user-defined types is described:

```
type, public, bind(c) :: full_particle !$fdps FP,EPI,EPJ,Force
```

FDPS must know which member variable of **FullParticle** type corresponds to which necessary quantity, where **necessary quantities** are defined as the quantities that are necessary in any types of particle simulations (e.g. mass (or charge) and position of a particle), or that are necessary in particular types of particle simulations (e.g. size of a particle). This designation is also done by adding a comment text with a special format to each member variable. In this sample code, in order to specify that member variables `mass`, `pos`, `vel` correspond to mass, position, velocity of a particle, the following directives are described:

```
real(kind=c_double) :: mass !$fdps charge
type(fdps_f64vec) :: pos !$fdps position
type(fdps_f64vec) :: vel !$fdps velocity
```

Note that `velocity` in the directive `!$fdps velocity` is a just reserved keyword and it does not alter the operation of FDPS at the present moment (hence, the designation is arbitrary).

FDPS copies data from **FullParticle** type to **EssentialParticleI** type and **EssentialParticleJ** type, or from **Force** type to **FullParticle** type. Users must describe FDPS directives that specify how to copy data. In this sample code, the following directives are described:

```
!$fdps copyFromForce full_particle (pot,pot) (acc,acc)
!$fdps copyFromFP full_particle (id,id) (mass,mass) (pos,pos)
```

where the FDPS directive with the keyword `copyFromForce` specifies which member variable of **Force** type is copied to which member variable of **FullParticle** type. Users always have to describe this directive in **FullParticle** type. The other directive with the keyword `copyFromFP` specifies how to copy data from **FullParticle** type to **EssentialParticleI** type and **EssentialParticleJ** type. This directive must always be described in **EssentialParticleI** type and **EssentialParticleJ** type. It is described here because **FullParticle** type in this sample code acts as **EssentialParticleI** type and **EssentialParticleJ** type.

**FullParticle** type also acts as **Force** type in this code. There is a FDPS directive that users must describe in **Force** type. It is the directive that specifies how to reset or initialize member variables of **Force** type before the calculation of interactions. In this code, the following directive is described to direct FDPS to zero-clear member variables corresponding to acceleration and potential only.

```
!$fdps clear id=keep, mass=keep, pos=keep, vel=keep
```

where the syntax `mbr=keep` to the right of the keyword `clear` is the syntax to direct FDPS not to change the value of member variable `mbr`.

Further details about the format of FDPS directive can be found in the specification document of FDPS Fortran/C interface, `doc_specs_ftn_en.pdf`.

#### 4.1.2.2 calcForceEpEp

You must define an interaction function `calcForceEpEp` as subroutine in Fortran. It should contain actual code for the calculation of interaction between particles. Listing 2 shows the implementation of `calcForceEpEp` (see `user_defined.F90` ).

Listing 2: Function `calcForceEpEp`

---

```

1  subroutine calc_gravity_ep_ep(ep_i,n_ip,ep_j,n_jp,f) bind(c)
2    implicit none
3    integer(c_int), intent(in), value :: n_ip,n_jp
4    type(full_particle), dimension(n_ip), intent(in) :: ep_i
5    type(full_particle), dimension(n_jp), intent(in) :: ep_j
6    type(full_particle), dimension(n_ip), intent(inout) :: f
7    !* Local variables
8    integer(c_int) :: i,j
9    real(c_double) :: eps2,poti,r3_inv,r_inv
10   type(fdps_f64vec) :: xi,ai,rij
11
12   !* Compute force
13   eps2 = eps_grav * eps_grav
14   do i=1,n_ip
15     xi = ep_i(i)%pos
16     ai = 0.0d0
17     poti = 0.0d0
18     do j=1,n_jp
19       rij%x = xi%x - ep_j(j)%pos%x
20       rij%y = xi%y - ep_j(j)%pos%y
21       rij%z = xi%z - ep_j(j)%pos%z
22       r3_inv = rij%x*rij%x +
23                 + rij%y*rij%y +
24                 + rij%z*rij%z +
25                 + eps2
26       r_inv = 1.0d0/sqrt(r3_inv)
27       r3_inv = r_inv * r_inv
28       r_inv = r_inv * ep_j(j)%mass
29       r3_inv = r3_inv * r_inv
30       ai%x = ai%x - r3_inv * rij%x
31       ai%y = ai%y - r3_inv * rij%y
32       ai%z = ai%z - r3_inv * rij%z
33       poti = poti - r_inv
34       ! [IMPORTANT NOTE]
35       ! In the innermost loop, we use the components of vectors
36       ! directly for vector operations because of the following
37       ! reason. Except for intel compilers with '-ipo' option,
38       ! most of Fortran compilers use function calls to perform
39       ! vector operations like rij = x - ep_j(j)%pos.
40       ! This significantly slow downs the speed of the code.
41       ! By using the components of vector directly, we can avoid
42       ! these function calls.
43     end do
44     f(i)%pot = f(i)%pot + poti
45     f(i)%acc = f(i)%acc + ai
46   end do
47
48 end subroutine calc_gravity_ep_ep

```

---

In this sample code, it is implemented as the subroutine `calc_gravity_ep_ep`. Its dummy arguments are an array of `EssentialParticleI` type, the number of `EssentialParticleI` type variables, an array of `EssentialParticleJ` type, the number of `EssentialParticleJ` type variables, an array of `Force` type. Note that all the data types of the dummy arguments corresponding to user-defined types are `full_particle` type because `FullParticle` type acts as the other types of user-defined types in this sample code.

#### 4.1.2.3 calcForceEpSp

You must defined an interaction function `calcForceEpSp` as subroutine in Fortran. It should contain actual code for the calculation of interaction between a particle and a super-particle. Listing 3 shows the implementation of `calcForceEpSp` (see `user_defined.F90`).

Listing 3: `calcForceEpSp`

---

```

1  subroutine calc_gravity_ep_sp(ep_i,n_ip,ep_j,n_jp,f) bind(c)
2    implicit none
3    integer(c_int), intent(in), value :: n_ip,n_jp
4    type(full_particle), dimension(n_ip), intent(in) :: ep_i
5    type(fdps_spj_monopole), dimension(n_jp), intent(in) :: ep_j
6    type(full_particle), dimension(n_ip), intent(inout) :: f
7    !* Local variables
8    integer(c_int) :: i,j
9    real(c_double) :: eps2,poti,r3_inv,r_inv
10   type(fdps_f64vec) :: xi,ai,rij
11
12   eps2 = eps_grav * eps_grav
13   do i=1,n_ip
14     xi = ep_i(i)%pos
15     ai = 0.0d0
16     poti = 0.0d0
17     do j=1,n_jp
18       rij%x = xi%x - ep_j(j)%pos%x
19       rij%y = xi%y - ep_j(j)%pos%y
20       rij%z = xi%z - ep_j(j)%pos%z
21       r3_inv = rij%x*rij%x &
22         + rij%y*rij%y &
23         + rij%z*rij%z &
24         + eps2
25       r_inv = 1.0d0/sqrt(r3_inv)
26       r3_inv = r_inv * r_inv
27       r_inv = r_inv * ep_j(j)%mass
28       r3_inv = r3_inv * r_inv
29       ai%x = ai%x - r3_inv * rij%x
30       ai%y = ai%y - r3_inv * rij%y
31       ai%z = ai%z - r3_inv * rij%z
32       poti = poti - r_inv
33     end do
34     f(i)%pot = f(i)%pot + poti
35     f(i)%acc = f(i)%acc + ai
36   end do
37
38 end subroutine calc_gravity_ep_sp

```

---

In this sample code, it is implemented as the subroutine `calc_gravity_ep_sp`. Its dummy arguments are an array of `EssentialParticle` type, the number of `EssentialParticle` type variables, an array of superparticle type, the number of superparticle type variables, an array of `Force` type. Note that the data types of `EssentialParticle` type and `Force` type are `full_particle` type because `FullParticle` type acts as these user-defined types in this sample code. Also note that the data type of superparticle type must be consistent with the type of a `Tree` object used in the calculation of interactions.

#### 4.1.3 The main body of the user program

In this section, we describe the functions a user should write in a kind of main routine, `f_main()`, to implement gravitational  $N$ -body calculation using the FDPS Fortran interface. The reason why we do not use the term main routine clearly is as follows: If users use FDPS Fortran interface, the user code must be written in the subroutine `f_main()`. Thus the user code dose not include the main routine or main program. However, in practice, the `f_main()` plays the same role as a main routine. Thus here we use the term a kind of main routine. The term main routine is suitable for indicating the top level function of the user code. Hereafter, we call `f_main()` the main routine. The main routine of this sample is written in `f_main.F90`.

##### 4.1.3.1 Creation of an object of type `fdps_controller`

In the FDPS Fortran interface, all APIs of FDPS are provided as member functions in the class `FDPS_controller`. This class is defined in the module `fdps_module` in `FDPS_module.F90`. Thus, in order to use APIs, the user must create an object of type `FDPS_controller`. In this sample, the object of type `FDPS_controller`, `fdps_ctrl`, is created in the main routine. Thus, in the following examples, APIs of FDPS are called as a member function of this object.

---

Listing 4: Creation of an object of type `fdps_controller`

---

```
1 subroutine f_main()
2   use fdps_module
3   implicit none
4   !* Local variables
5   type(fdps_controller) :: fdps_ctrl
6
7   ! Do something
8
9 end subroutine f_main
```

---

Note that the code shown above is an only necessary part from the sample code.

##### 4.1.3.2 Initialization and Termination of FDPS

First, users must initialize FDPS by the following code.

---

Listing 5: Initialization of FDPS

---

```
1 call fdps_ctrl%PS_Initialize()
```

---

Once started, FDPS should be terminated explicitly. In the sample code, FDPS should be terminated just before the termination of the program. To achieve this, user should write the following code at the end of the main routine.

Listing 6: Termination of FDPS

---

```
1 call fdps_ctrl%ps_finalize()
```

---

#### 4.1.3.3 Creation and initialization of FDPS objects

Once succeed the initialization, the user needs to create objects used to talk to FDPS. In this section, we describe how to create and initialize these objects.

##### 4.1.3.3.1 Creation of FDPS objects

In an  $N$ -body simulation, one needs to create objects of `ParticleSystem` type, `DomainInfo` type, and `Tree` type. In the Fortran interface, these objects can be handled by using identification number contained in integral type variables. Thus, at the beginning, you should prepare integral type variables to contain the identification numbers. We will show an example bellow. These are written in the main routine `f_main.F90` in the sample code.

Listing 7: Creation of FDPS objects

---

```
1 subroutine f_main()
2   use fdps_module
3   use user_defined_types
4   implicit none
5   !* Local variables
6   integer :: psys_num, dinfo_num, tree_num
7
8   !* Create FDPS objects
9   call fdps_ctrl%create_dinfo(dinfo_num)
10  call fdps_ctrl%create_psyst(psys_num, 'full_particle')
11  call fdps_ctrl%create_tree(tree_num, &
12    "Long,full_particle,full_particle,
13      full_particle,Monopole")
14 end subroutine f_main
```

---

Here, the code shown is just a corresponding part of the sample code. As we can see above, to create the object of type `ParticleSystem`, you must give the string of the name of the derived data type corresponding to the type `FullParticle`. As in the case of type `ParticleSystem`, to create the object of type `Tree`, you must give the string which indicates the type of tree as an argument of the API. Note that, in both APIs, the name of the derived data type must be written in lower case.

##### 4.1.3.3.2 Initialization of `DomainInfo` object

Once create the objects, user must initialize these objects. In this sample code, since the boundary condition is not periodic, users have only to call the API `init_dinfo` to initialize the objects.

Listing 8: Initialization of DomainInfo object

---

```
1 call fdps_ctrl%init_dinfo(dinfo_num,coef_ema)
```

---

Note that the second argument of API `init_dinfo` is a smoothing factor of an exponential moving average operation that is performed in the domain decomposition procedure. The definition of this factor is described in the specification of FDPS (see § 9.1.2 in `doc_spec_cpp_en.pdf`).

#### 4.1.3.3.3 Initialization of ParticleSystem object

Next, you must initialize a `ParticleSystem` object. This is done by calling the API `init_psys`.

Listing 9: Initialization of ParticleSystem object

---

```
1 call fdps_ctrl%init_psys(psyst_num)
```

---

#### 4.1.3.3.4 Initialization of Tree object

Next, we must initialize a `Tree` object. The initialization of a `Tree` object is done by calling the API `init_tree`. This API should be given an integer for determining the initial sizes of internally-used arrays. A number comparable to the number of local particles is enough. In this sample, we set the number of local particles (`n_loc`):

Listing 10: Initialization of Tree object

---

```
1 call fdps_ctrl%init_tree(tree_num,n_loc,theta,&
2 n_leaf_limit,n_group_limit)
```

---

The `initialize` method has three optional arguments. Here, we pass these arguments explicitly.

- `theta` — the so-called opening angle criterion for the tree method.
- `n_leaf_limit` — the upper limit for the number of particles in the leaf nodes.
- `n_group_limit` — the upper limit for the number of particles with which the particles use the same interaction list for the force calculation.

#### 4.1.3.4 Initialization of particle data

To initialize particle data, users must give the particle data to the `ParticleSystem` object. This can be done by using APIs `set_nptcl_loc` and `get_psyst_fptr` as follows:

Listing 11: Initialization of particle data

---

```
1 subroutine foo(fdps_ctrl,psyst_num)
2   use fdps_vector
3   use fdps_module
4   use user_defined_types
5   implicit none
6   type(fdps_controller), intent(IN) :: fdps_ctrl
7   integer, intent(IN) :: psyst_num
8   !* Local variables
9   integer :: i,nptcl_loc
```

---

---

```

10 type(full_particle), dimension(:), pointer :: ptcl
11
12 !* Set # of local particles
13 call fdps_ctrl%set_nptcl_loc(psys_num,nptcl_loc)
14
15 !* Get the pointer to full particle data
16 call fdps_ctrl%get_psyst_fptr(psys_num,ptcl)
17
18 !* Initialize particle data
19 do i=1,nptcl_loc
20     ptcl(i)%pos = ! Do something
21 end do
22
23 !* Release the pointer
24 nullify(ptcl)
25
26 end subroutine foo

```

---

First, you must allocate the memory to store the particle data. To do so, you have only to call API `set_nptcl_loc`. This API sets the number of local particles (the number of particles assigned to the local process) and allocates enough memory to store the particles. To initialize particle data, the beginning address of the allocated memory is needed. Users can obtain the beginning address by using the API `get_psyst_fptr`. Users must receive the beginning address by a Fortran pointer. In the example above, the pointer is prepared as follows:

```
type(full_particle), dimension(:), pointer :: ptcl
```

Once you sets the pointer by the API `get_psyst_fptr`, you can use the pointer as an array. In the above example, after initialize particle data, the pointer is freed by the built-in function `nullify`.

#### 4.1.3.5 Time integration loop

In this section we describe the structure of the time integration loop.

##### 4.1.3.5.1 Domain Decomposition

First, the computational domain is decomposed, using the current distribution of particles. In the sample, this is done by API `decompose_domain_all` of the `DomainInfo` object:

Listing 12: Domain Decomposition

---

```

1 if (mod(num_loop,4) == 0) then
2     call fdps_ctrl%decompose_domain_all(dinfo_num,psys_num)
3 end if

```

---

In this sample code, we perform domain decomposition once in 4 main loops in order to reduce the computational cost.

#### 4.1.3.5.2 Particle Exchange

Then, particles are exchanged between processes so that they belong to the process for the domain of their coordinates. To do so, users can use API `exchange_particle` of `ParticleSystem` object.

Listing 13: Particle Exchange

---

```
1 call fdps_ctrl%exchange_particle(psys_num, dinfo_num)
```

---

#### 4.1.3.5.3 Interaction Calculation

After the domain decomposition and the particle exchange, an interaction calculation is done. To do so, users can use API `calc_force_all_and_write_back` of `Tree` object.

Listing 14: Interaction Calculation

---

```
1 subroutine f_main()
2   use, intrinsic :: iso_c_binding
3   use user_defined_types
4   implicit none
5   !* Local variables
6   type(c_funptr) :: pfunc_ep_ep, pfunc_ep_sp
7
8   ! Do something
9
10  pfunc_ep_ep = c_funloc(calc_gravity_pp)
11  pfunc_ep_sp = c_funloc(calc_gravity_psp)
12  call fdps_ctrl%calc_force_all_and_write_back(tree_num,      &
13                                              pfunc_ep_ep,      &
14                                              pfunc_ep_sp,      &
15                                              psys_num,        &
16                                              dinfo_num)
17
18   ! Do something
19
20 end subroutine f_main
```

---

Here, the second and the third arguments are functions pointers of `calcForceEpEp` and `calcForceEpSp`. The address of the function in C can be obtained using the built-in function `c_funloc`, which is introduced in Fortran 2003. This built-in function is provided by the module `iso_c_binding` and we use `use` statement to use this module. To store the address in C, we need the variables of derived data type `c_funptr`, which is also introduced in Fortran 2003. In this sample, we use variables of type `c_funptr`, `pfunc_ep_ep` and `pfunc_ep_sp`, to store the address in C of `calc_gravity_pp` and `calc_gravity_psp` and give them to the API.

#### 4.1.3.5.4 Time integration

In this sample code, we use the Leapfrog method to integrate the particle system in time. In this method, the time evolution operator can be expressed as  $K(\frac{\Delta t}{2})D(\Delta t)K(\frac{\Delta t}{2})$ , where  $\Delta t$  is the timestep,  $K(\Delta t)$  is the ‘kick’ operator that integrates the velocities of particles

from  $t$  to  $t + \Delta t$ ,  $D(\Delta t)$  is the ‘drift’ operator that integrates the positions of particles from  $t$  to  $t + \Delta t$  (e.g. see Springel [2005,MNRAS,364,1105]). In the sample code, these operators are implemented as the subroutines `kick` and `drift`.

At the beginning of the main loop, the positions and the velocities of the particles are updated by the operator  $D(\Delta t)K(\frac{\Delta t}{2})$ :

---

Listing 15: Calculation of  $D(\Delta t)K(\frac{\Delta t}{2})$ operator

---

```

1 /* Leapfrog: Kick-Drift
2 call kick(fdps_ctrl,psys_num,0.5d0*dt)
3 time_sys = time_sys + dt
4 call drift(fdps_ctrl,psys_num,dt)
```

---



---

Listing 16: Calculation of  $D(\Delta t)K(\frac{\Delta t}{2})$ operator

---

```

1 // Leapfrog: Kick-Drift
2 kick(psys_num,0.5*dt);
3 time_sys += dt;
4 drift(psys_num,dt);
```

---

After the force calculation, the velocities of the particles are updated by the operator  $K(\frac{\Delta t}{2})$ :

---

Listing 17: Calculation of  $K(\frac{\Delta t}{2})$  operator

---

```

1 /* Leapfrog: Kick
2 call kick(fdps_ctrl,psys_num,0.5d0*dt)
```

---

#### 4.1.3.6 Update of particle data

To update the data of particles in the subroutines such as `kick` or `drift`, you need to access the data of particles contained in the object of type `ParticleSystem`. To do so, the user can follow the same way described in section 4.1.3.4.

---

Listing 18: Update of particle data

---

```

1 subroutine foo(fdps_ctrl,psys_num)
2   use fdps_vector
3   use fdps_module
4   use user_defined_types
5   implicit none
6   type(fdps_controller), intent(IN) :: fdps_ctrl
7   integer, intent(IN) :: psys_num
8   !* Local variables
9   integer :: i,nptcl_loc
10  type(full_particle), dimension(:), pointer :: ptcl
11
12  !* Get # of local particles
13  nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
14
15  !* Get the pointer to full particle data
16  call fdps_ctrl%get_psyst_fptr(psys_num,ptcl)
17
18  !* Initialize or update particle data
19  do i=1,nptcl_loc
```

---

```

20      ptcl(i)%pos = ! Do something
21  end do
22
23  /* Release the pointer
24  nullify(ptcl)
25
26 end subroutine foo

```

---

Using API `get_psyst_fptr`, you can obtain the address of particle data contained in the object of `ParticleSystem` as a pointer. The pointer obtained here can be regarded as an array with the size of `nptcl_loc`. Thus user can update the particle data as array.

#### 4.1.4 Log file

Once the calculation starts successfully, the time and the energy error are printed in the standard output. The first step is shown in the bellow example.

Listing 19: standard output

---

```

1 time: 0.0000000000E+000, energy error: -0.0000000000E+000

```

---

## 4.2 SPH simulation code with fixed smoothing length

In this section, we describe the sample code used in the previous section (§ 3), a standard SPH code with fixed smoothing length, in detail.

### 4.2.1 Location of source files and file structure

The source files of the sample code are in the directory `$(FDPS)/sample/fortran/sph`. The sample code consists of `user_defined.F90` where user-defined types are described, and `f_main.F90` where the main loop etc. of the SPH simulation code are described. In addition, there are two Makefiles: `Makefile` (for GCC) and `Makefile.intel` (for Intel compilers).

### 4.2.2 User-defined types and user-defined functions

In this section, we describe the derived data types and subroutines that users must define when performing SPH simulations by using of FDPS.

#### 4.2.2.1 FullParticle type

Users must define a `FullParticle` type as a user-defined type. The `FullParticle` type must contain all physical quantities of an SPH particle necessary for the simulation. Listing 20 shows an example implementation of the `FullParticle` type in our sample code (see `user_defined.F90` ).

Listing 20: FullParticle type

---

```

1 !**** Full particle type
2 type, public, bind(c) :: full_particle !$fdps FP
3   !$fdps copyFromForce force_dens (dens,dens)
4   !$fdps copyFromForce force_hydro (acc,acc) (eng_dot,eng_dot) (dt,dt)

```

---

---

```

5      real(kind=c_double) :: mass !$fdps charge
6      type(fdps_f64vec) :: pos !$fdps position
7      type(fdps_f64vec) :: vel
8      type(fdps_f64vec) :: acc
9      real(kind=c_double) :: dens
10     real(kind=c_double) :: eng
11     real(kind=c_double) :: pres
12     real(kind=c_double) :: smth !$fdps rsearch
13     real(kind=c_double) :: snds
14     real(kind=c_double) :: eng_dot
15     real(kind=c_double) :: dt
16     integer(kind=c_long_long) :: id
17     type(fdps_f64vec) :: vel_half
18     real(kind=c_double) :: eng_half
19   end type full_particle

```

---

Unlike the case of the *N*-body simulation sample code, the **FullParticle** type of the SPH simulation sample code does not double as other user-defined types. Thus, to specify that this derived data type is a **FullParticle** type, we append the following directive.

```
type, public, bind(c) :: full_particle !$fdps FP
```

In the SPH simulations, the interaction force is short-range force. Therefore, a search radius is also necessary physical quantity in addition to the position and mass (charge). We can tell FDPS which member variables represent these necessary quantities in the following way:

```
real(kind=c_double) :: mass !$fdps charge
type(fdps_f64vec) :: pos !$fdps position
real(kind=c_double) :: smth !$fdps rsearch
```

As described in the section of the *N*-body simulation code, the keyword **velocity** to specify that a member corresponds to the velocity of a particle is mere a reserved word and not always necessary, we do not specify that in this sample code.

The **FullParticle** type copies data from a **Force** type. Users must specify how the data is copied by using of directives. As we will describe later, there are 2 **Force** types in this SPH sample code. Thus, for each **Force** type, users must write the directives. In this sample code, these are:

```
!$fdps copyFromForce force_dens (dens,dens)
!$fdps copyFromForce force_hydro (acc,acc) (eng_dot,eng_dot) (dt,dt)
```

#### 4.2.2.2 EssentialParticleI(J) type

Users must define an **EssentialParticleI** type. An **EssentialParticleI** type must contain all necessary physical quantities to compute the **Force** as an *i*-particle in its member variables. Moreover in this sample code, it also doubles as an **EssentialParticleJ** type and all necessary physical quantities as a *j*-particle as well need to be included in the member variables. Hereinafter, we simply call this **EssentialParticle** type. Listing 21 shows an example of **EssentialParticle** type of this sample code (see `user_defined.F90`):

Listing 21: EssentialParticle type

---

```

1  **** Essential particle type
2  type, public, bind(c) :: essential_particle !$fdps EPI,EPJ
3      !$fdps copyFromFP full_particle (id,id) (pos,pos) (vel,vel) (mass,
4          mass) (smth,smth) (dens,dens) (pres,pres) (sns,snds)
5      integer(kind=c_long_long) :: id !$fdps id
6      type(fdps_f64vec) :: pos !$fdps position
7      type(fdps_f64vec) :: vel
8      real(kind=c_double) :: mass !$fdps charge
9      real(kind=c_double) :: smth !$fdps rsearch
10     real(kind=c_double) :: dens
11     real(kind=c_double) :: pres
12     real(kind=c_double) :: sns
13 end type essential_particle

```

---

First, users must indicate to FDPS that this derived data type corresponds to both the `EssentialParticleI` type and `EssentialParticleJ` type by using the directives. This sample code describes that as follows:

```
type, public, bind(c) :: essential_particle !$fdps EPI,EPJ
```

Next, users must indicate the correspondence between the each of member variable in this derived data type and necessary physical quantity. For this SPH simulation, a search radius needs to be indicated as well. This sample code describes them as follows:

```
type(fdps_f64vec) :: pos !$fdps position
real(kind=c_double) :: mass !$fdps charge
real(kind=c_double) :: smth !$fdps rsearch
```

The `EssentialParticleI` and `EssentialParticleJ` types receive data from the `FullParticle` type. Users must specify the source member variables in the `FullParticle` type and the destination member variable in the `EssentialParticle?` type (?=I,J) that will be copied through the directives. This sample code describes them as follows:

```
!$fdps copyFromFP full_particle (id,id) (pos,pos) (vel,vel) (mass,mass)
(smth,smth) (dens,dens) (pres,pres) (sns,snds)
```

#### 4.2.2.3 Force type

Users must define a `Force` type. A `Force` type must contain all the resultant physical quantities after performing the `Force` computations. In this sample code, we have 2 force computations; one for the density and the other for the fluid interactions. Thus, we have to define 2 different `Force` types. In Listing 22, we show an example of the `Force` types in this sample code.

Listing 22: Force type

---

```

1  **** Force types
2  type, public, bind(c) :: force_dens !$fdps Force
3      !$fdps clear smth=keep

```

---

```

4      real(kind=c_double) :: dens
5      real(kind=c_double) :: smth
6  end type force_dens
7
8  type, public, bind(c) :: force_hydro !$fdps Force
9    !$fdps clear
10   type(fdps_f64vec) :: acc
11   real(kind=c_double) :: eng_dot
12   real(kind=c_double) :: dt
13 end type force_hydro

```

First, users must indicate with directives that these derived data types correspond to the **Force** types. In this example, these writes:

```

type, public, bind(c) :: force_dens !$fdps Force
type, public, bind(c) :: force_hydro !$fdps Force

```

For these derived data types to be **Force** types, users **must** indicate the initialization methods for the member variables that are accumulated during the interaction calculations. In this sample code, we indicate that only the accumulator variables — density, acceleration (due to pressure gradient), time-derivative of energy, and time step to be zero-cleared.

```

!$fdps clear smth=keep
!$fdps clear

```

In this example the **Force** type **force\_dens** includes a member variable **smth** that indicates the smoothing length. For a fixed length SPH, a member variable for the smoothing length in the **Force** type has nothing to do. We prepare this member variable for the future extension to the variable length SPH for some users. In one of the formulations of the variable length SPH in Springel [2005,MNRAS,364,1105], we need to calculate the smoothing length at the same time we calculate the density. To implement a formulation like that, a **Force** type need to contain a variable for the smoothing length as in this example. In this sample code for fixed length SPH, the member function **clear** will not zero-clear the variable **smth**, so as not to crush the next computation of the density.

#### 4.2.2.4 calcForceEpEp

Users must define a subroutine **calcForceEpEp** in Fortran which specifies the interaction between particles. It should contain actual code for the calculation of interaction between particles. Listing 23 shows the implementation of **calcForceEpEp** (see `user_defined.c`).

Listing 23: Function calcForceEpEp

```

1  **** Interaction function
2  subroutine calc_density(ep_i,n_ip,ep_j,n_jp,f) bind(c)
3    integer(kind=c_int), intent(in), value :: n_ip,n_jp
4    type(essential_particle), dimension(n_ip), intent(in) :: ep_i
5    type(essential_particle), dimension(n_jp), intent(in) :: ep_j
6    type(force_dens), dimension(n_ip), intent(inout) :: f
7    !* Local variables
8    integer(kind=c_int) :: i,j

```

```

9      type(fdps_f64vec) :: dr
10
11     do i=1,n_ip
12       f(i)%dens = 0.0d0
13       do j=1,n_jp
14         dr%x = ep_j(j)%pos%x - ep_i(i)%pos%x
15         dr%y = ep_j(j)%pos%y - ep_i(i)%pos%y
16         dr%z = ep_j(j)%pos%z - ep_i(i)%pos%z
17         f(i)%dens = f(i)%dens &
18             + ep_j(j)%mass * W(dr,ep_i(i)%smth)
19       end do
20     end do
21
22   end subroutine calc_density
23
24 !**** Interaction function
25 subroutine calc_hydro_force(ep_i,n_ip,ep_j,n_jp,f) bind(c)
26   integer(kind=c_int), intent(in), value :: n_ip,n_jp
27   type(essential_particle), dimension(n_ip), intent(in) :: ep_i
28   type(essential_particle), dimension(n_jp), intent(in) :: ep_j
29   type(force_hydro), dimension(n_ip), intent(inout) :: f
30   !* Local parameters
31   real(kind=c_double), parameter :: C_CFL=0.3d0
32   !* Local variables
33   integer(kind=c_int) :: i,j
34   real(kind=c_double) :: mass_i,mass_j,smth_i,smth_j, &
35                         dens_i,dens_j,pres_i,pres_j, &
36                         snds_i,snds_j
37   real(kind=c_double) :: povrho2_i,povrho2_j, &
38                         v_sig_max,dr_dv,w_ij,v_sig,AV
39   type(fdps_f64vec) :: pos_i,pos_j,vel_i,vel_j, &
40                         dr,dv,gradW_ij
41
42   do i=1,n_ip
43     !* Zero-clear
44     v_sig_max = 0.0d0
45     !* Extract i-particle info.
46     pos_i = ep_i(i)%pos
47     vel_i = ep_i(i)%vel
48     mass_i = ep_i(i)%mass
49     smth_i = ep_i(i)%smth
50     dens_i = ep_i(i)%dens
51     pres_i = ep_i(i)%pres
52     snds_i = ep_i(i)%snds
53     povrho2_i = pres_i/(dens_i*dens_i)
54     do j=1,n_jp
55       !* Extract j-particle info.
56       pos_j%x = ep_j(j)%pos%x
57       pos_j%y = ep_j(j)%pos%y
58       pos_j%z = ep_j(j)%pos%z
59       vel_j%x = ep_j(j)%vel%x
60       vel_j%y = ep_j(j)%vel%y
61       vel_j%z = ep_j(j)%vel%z
62       mass_j = ep_j(j)%mass
63       smth_j = ep_j(j)%smth

```

```

64      dens_j = ep_j(j)%dens
65      pres_j = ep_j(j)%pres
66      snds_j = ep_j(j)%snds
67      povrho2_j = pres_j/(dens_j*dens_j)
68      !* Compute dr & dv
69      dr%x = pos_i%x - pos_j%x
70      dr%y = pos_i%y - pos_j%y
71      dr%z = pos_i%z - pos_j%z
72      dv%x = vel_i%x - vel_j%x
73      dv%y = vel_i%y - vel_j%y
74      dv%z = vel_i%z - vel_j%z
75      !* Compute the signal velocity
76      dr_dv = dr%x * dv%x + dr%y * dv%y + dr%z * dv%z
77      if (dr_dv < 0.0d0) then
78          w_ij = dr_dv / sqrt(dr%x * dr%x + dr%y * dr%y + dr%z * dr%z)
79      )
80      else
81          w_ij = 0.0d0
82      end if
83      v_sig = snds_i + snds_j - 3.0d0 * w_ij
84      v_sig_max = max(v_sig_max, v_sig)
85      !* Compute the artificial viscosity
86      AV = - 0.5d0*v_sig*w_ij / (0.5d0*(dens_i+dens_j))
87      !* Compute the average of the gradients of kernel
88      gradW_ij = 0.5d0 * (gradW(dr,smth_i) + gradW(dr,smth_j))
89      !* Compute the acceleration and the heating rate
90      f(i)%acc%x = f(i)%acc%x - mass_j*(povrho2_i+povrho2_j+AV)*
91          gradW_ij%x
92      f(i)%acc%y = f(i)%acc%y - mass_j*(povrho2_i+povrho2_j+AV)*
93          gradW_ij%y
94      f(i)%acc%z = f(i)%acc%z - mass_j*(povrho2_i+povrho2_j+AV)*
95          gradW_ij%z
96      f(i)%eng_dot = f(i)%eng_dot &
97          + mass_j * (povrho2_i + 0.5d0*AV) &
98              *(dv%x * gradW_ij%x &
99                  +dv%y * gradW_ij%y &
100                     +dv%z * gradW_ij%z)
101     end do
102     f(i)%dt = C_CFL*2.0d0*smth_i/(v_sig_max*kernel_support_radius)
103 end do
104 ! [IMPORTANT NOTE]
105 ! In the innermost loop, we use the components of vectors
106 ! directly for vector operations because of the following
107 ! reasion. Except for intel compilers with ‘-ipo’ option,
108 ! most of Fortran compilers use function calls to perform
109 ! vector operations like rij = x - ep_j(j)%pos.
110 ! This significantly slow downs the speed of the code.
111 ! By using the components of vector directly, we can avoid
112 ! these function calls.
113
114 end subroutine calc_hydro_force

```

This SPH simulation code includes two different forms of interactions, and hence, two different implementations of `calcForceEpEp` are needed. In either case, the dummy arguments of subroutine are, an array of `EssentialParticle`, the number of `EssentialParticle`, an array of

`EssentialParticleJ`, the number of `EssentialParticleJ`, and an array of `Force`.

### 4.2.3 The main body of the user program

In this section, we describe subroutines and functions to be called from the main routine of the user program when a user want to do an SPH simulation using FDPS (for the meaning of “main routine” see section 4.1.3) .

#### 4.2.3.1 Creation of an object of type `fdps_controller`

In order to use APIs of FDPS, a user program should create an object of type `FDPS_controller`. In this sample code, `fdps_ctrl`, an object of type `FDPS_controller`, is created in the main routine.

Listing 24: Creation of an object of type `fdps_controller`

---

```
1 subroutine f_main()
2   use fdps_module
3   implicit none
4   !* Local variables
5   type(fdps_controller) :: fdps_ctrl
6
7   ! Do something
8
9 end subroutine f_main
```

---

Note that this code snippet only shows the necessary part of the code from the actual sample code. Also note that all FDPS APIs are called as member functions of this object because of the reason described above.

#### 4.2.3.2 Initialization and termination of FDPS

You should first initialize FDPS by the following code.

Listing 25: Initialization of FDPS

---

```
1 call fdps_ctrl%PS_Initialize()
```

---

Once started, FDPS should be explicitly terminated. In this sample, FDPS is terminated just before the termination of the program. To achieve this, you write the following code at the end of the main routine.

Listing 26: Termination of FDPS

---

```
1 call fdps_ctrl%PS_Finalize()
```

---

#### 4.2.3.3 Creation and initialization of FDPS objects

After the initialization of FDPS, a user need to create the objects used to talk to FDPS. In this section we describe how to create and initialize these objects.

#### 4.2.3.3.1 Creation of necessary FDPS objects

In an SPH simulation code, one needs to create objects for particles, for domain information, for interaction calculation of Gather type (for density calculation using gather type interaction), and for interaction calculation of Symmetry type (for hydrodynamic interaction calculation using symmetric type interaction). The following is the code to create to them.

Listing 27: Creation of necessary FDPS objects

---

```

1 subroutine f_main()
2   use fdps_vector
3   use fdps_module
4   use user_defined_types
5   implicit none
6   !* Local variables
7   integer :: psys_num, dinfo_num
8   integer :: dens_tree_num, hydro_tree_num
9
10  !* Create FDPS objects
11  call fdps_ctrl%create_psys(psys_num, 'full_particle')
12  call fdps_ctrl%create_dinfo(dinfo_num)
13  call fdps_ctrl%create_tree(dens_tree_num, &
14                           "Short, dens_force, essential_particle,
15                           essential_particle, Gather")
16  call fdps_ctrl%create_tree(hydro_tree_num, &
17                           "Short, hydro_force, essential_particle,
18                           essential_particle, Symmetry")
18 end subroutine f_main

```

---

Note that here again this code snippet only shows the necessary part of the code from the actual sample code.

API `create_psys` and `create_tree` should receive strings indicating particle type and tree type, respectively. **All of names of derived data types in these strings should be in lowercases.**

#### 4.2.3.3.2 Initialization of the domain information object

FDPS objects created by a user code should be initialized. Here, we describe the necessary procedures required to initialize a `DomainInfo` object. First, we need to call API `init_dinfo` of `DomainInfo` object. After the initialization of the object, the type of the boundary and the size of the simulation box should be set by calling APIs `set_boundary_condition` and `set_pos_root_domain` of `DomainInfo` object. In this code, we use the periodic boundary for all of  $x$ ,  $y$  and  $z$  directions.

Listing 28: Initialization of DomainInfo object

---

```

1 call fdps_ctrl%init_dinfo(dinfo_num, coef_ema)
2 call fdps_ctrl%set_boundary_condition(dinfo_num, fdps_bc_periodic_xyz)
3 call fdps_ctrl%set_pos_root_domain(dinfo_num, pos_ll, pos_ul)

```

---

#### 4.2.3.3.3 Initialization of ParticleSystem object

Next, we need to initialize the `ParticleSystem` object. This is done by the following single line of code:

---

Listing 29: Initialization of ParticleSystem object

---

```
1 call fdps_ctrl%init_psys(psys_num)
```

---

#### 4.2.3.3.4 Initialization of Tree objects

Finally, `Tree` objects should be initialized. This is done by calling API `init_tree`. This API should be given an integer for determining the initial sizes of internally-used arrays. A number comparable to the number of local particles is enough. In this sample, we set three times the number of local particles:

---

Listing 30: Initialization of tree objects

---

```
1 call fdps_ctrl%init_tree(dens_tree_num,n_loc,theta,&
2                               n_leaf_limit,n_group_limit)
3 call fdps_ctrl%init_tree(hydro_tree_num,n_loc,theta,&
4                               n_leaf_limit,n_group_limit)
```

---

#### 4.2.3.4 Time integration loop

In this section we describe the structure of the time integration loop.

##### 4.2.3.4.1 Domain Decomposition

First, the computational domain is decomposed, using the current distribution of particles. To do so, the API `decompose_domain_all` of `DomainInfo` object is called.

---

Listing 31: Domain Decomposition

---

```
1 call fdps_ctrl%decompose_domain_all(dinfo_num,psys_num)
```

---

##### 4.2.3.4.2 Particle Exchange

Then particles are exchanged between processes so that they belong to the process for the domain of their coordinates. To do so, the following API `exchange_particle` of `ParticleSystem` object is used.

---

Listing 32: Particle Exchange

---

```
1 call fdps_ctrl%exchange_particle(psys_num,dinfo_num)
```

---

##### 4.2.3.4.3 Interaction Calculation

After the domain decomposition and particle exchange, interaction calculation is done. To do so, the following API `calc_force_all_and_write_back` of `Tree` object is used.

Listing 33: Interaction Calculation

---

```

1 subroutine f_main()
2   use, intrinsic :: iso_c_binding
3   use user_defined_types
4   implicit none
5   !* Local variables
6   type(c_funptr) :: pfunc_ep_ep
7
8   ! Do something
9
10  pfunc_ep_ep = c_funloc(calc_density)
11  call fdps_ctrl%calc_force_all_and_write_back(dens_tree_num, &
12                                              pfunc_ep_ep,      &
13                                              psys_num,        &
14                                              dinfo_num)
15  call set_pressure(fdps_ctrl,psys_num)
16  pfunc_ep_ep = c_funloc(calc_hydro_force)
17  call fdps_ctrl%calc_force_all_and_write_back(hydro_tree_num, &
18                                              pfunc_ep_ep,      &
19                                              psys_num,        &
20                                              dinfo_num)
21
22   ! Do something
23
24 end subroutine f_main

```

---

For the second argument of API, the function pointer (as in the C language) of function calcForceEpEp should be given.

#### 4.2.4 Compilation of the program

Run `make` at the working directory. You can use the Makefile attached to the sample code.

```
$ make
```

#### 4.2.5 Execution

To run the code without MPI, you should execute the following command in the command shell.

```
$ ./sph.out
```

To run the code using MPI, you should execute the following command in the command shell, or follow the document of your system.

```
$ MPIRUN -np NPROC ./sph.out
```

Here, `MPIRUN` represents the command to run your program using MPI such as `mpirun` or `mpiexec`, and `NPROC` is the number of MPI processes.

#### 4.2.6 Log and output files

Log and output files are created under `result` directory.

#### 4.2.7 Visualization

In this section, we describe how to visualize the calculation result using `gnuplot`. To enter the interactive mode of `gnuplot`, execute the following command.

```
$ gnuplot
```

In the interactive mode, you can visualize the result. In the following example, using the 50th snapshot file, we create the plot in which the abscissa is the x coordinate of particles and the ordinate is the density of particles.

```
gnuplot> plot "result/snap00050-proc00000.dat" u 3:9
```

where the integral number after the string of characters `proc` represents the rank number of a MPI process.

# 5 Sample Codes

## 5.1 *N*-body simulation

In this section, we show a sample code for the *N*-body simulation. This code is the same as what we described in section 4. One can create a working code by cut and paste this code and compile and link the resulted source program.

Listing 34: Sample code of *N*-body simulation (`user_defined.F90`)

```

1 !=====
2 !   MODULE: User defined types
3 !=====
4 module user_defined_types
5   use, intrinsic :: iso_c_binding
6   use fdps_vector
7   use fdps_super_particle
8   implicit none
9
10  !* Public variables
11  real(kind=c_double), public :: eps_grav ! gravitational softening
12
13  !**** Full particle type
14  type, public, bind(c) :: full_particle !$fdps FP,EPI,EPJ,Force
15    !$fdps copyFromForce full_particle (pot,pot) (acc,acc)
16    !$fdps copyFromFP full_particle (id,id) (mass,mass) (pos,pos)
17    !$fdps clear id=keep, mass=keep, pos=keep, vel=keep
18    integer(kind=c_long_long) :: id
19    real(kind=c_double) mass !$fdps charge
20    type(fdps_f64vec) :: pos !$fdps position
21    type(fdps_f64vec) :: vel !$fdps velocity
22    real(kind=c_double) :: pot
23    type(fdps_f64vec) :: acc
24  end type full_particle
25
26  !* The following types are used in PIKG-generated kernels
27  type, public, bind(c) :: epi_grav
28    type(fdps_f32vec) :: pos
29  end type epi_grav
30
31  type, public, bind(c) :: epj_grav
32    type(fdps_f32vec) :: pos
33    real(kind=c_float) :: mass
34  end type epj_grav
35
36  type, public, bind(c) :: force_grav
37    type(fdps_f32vec) :: acc
38    real(kind=c_float) :: pot
39  end type force_grav
40
41  contains
42
43  !**** Interaction function (particle-particle)
44 #if defined(ENABLE_PHANTOM_GRAPE_X86)
45  subroutine calc_gravity_ep_ep(ep_i,n_ip,ep_j,n_jp,f) bind(c)

```

```

46 #if defined(PARTICLE_SIMULATOR_THREAD_PARALLEL) && defined(_OPENMP)
47     use omp_lib
48 #endiff
49     use phantom_grape_g5_x86
50     implicit none
51     integer(c_int), intent(in), value :: n_ip,n_jp
52     type(full_particle), dimension(n_ip), intent(in) :: ep_i
53     type(full_particle), dimension(n_jp), intent(in) :: ep_j
54     type(full_particle), dimension(n_ip), intent(inout) :: f
55     !* Local variables
56     integer(c_int) :: i,j
57     integer(c_int) :: npipe,npjpipe,devid
58     real(c_double), dimension(3,n_ip) :: xi,ai
59     real(c_double), dimension(n_ip) :: pi
60     real(c_double), dimension(3,n_jp) :: xj
61     real(c_double), dimension(n_jp) :: mj
62
63     npipe = n_ip
64     npjpipe = n_jp
65     do i=1,n_ip
66         xi(1,i) = ep_i(i)%pos%x
67         xi(2,i) = ep_i(i)%pos%y
68         xi(3,i) = ep_i(i)%pos%z
69         ai(1,i) = 0.0d0
70         ai(2,i) = 0.0d0
71         ai(3,i) = 0.0d0
72         pi(i) = 0.0d0
73     end do
74     do j=1,n_jp
75         xj(1,j) = ep_j(j)%pos%x
76         xj(2,j) = ep_j(j)%pos%y
77         xj(3,j) = ep_j(j)%pos%z
78         mj(j) = ep_j(j)%mass
79     end do
80 #if defined(PARTICLE_SIMULATOR_THREAD_PARALLEL) && defined(_OPENMP)
81     devid = omp_get_thread_num()
82     ! [IMPORTANT NOTE]
83     ! The subroutine calc_gravity_pp is called by a OpenMP thread
84     ! in the FDPS. This means that here is already in the parallel
85     ! region.
86     ! So, you can use omp_get_thread_num() without !$OMP parallel
87     ! directives.
88     ! If you use them, a nested parallel regions is made and the
89     ! gravity
90     ! calculation will not be performed correctly.
91 #else
92     devid = 0
93 #endiff
94     call g5_set_xmjMC(devid, 0, n_jp, xj, mj)
95     call g5_set_nMC(devid, n_jp)
96     call g5_calculate_force_on_xMC(devid, xi, ai, pi, n_ip)
97     do i=1,n_ip
98         f(i)%acc%x = f(i)%acc%x + ai(1,i)
99         f(i)%acc%y = f(i)%acc%y + ai(2,i)
100        f(i)%acc%z = f(i)%acc%z + ai(3,i)

```

```

98      f(i)%pot    = f(i)%pot    - pi(i)
99      end do
100     end subroutine calc_gravity_ep_ep
101
102     subroutine calc_gravity_ep_sp(ep_i,n_ip,ep_j,n_jp,f) bind(c)
103 #if defined(PARTICLE_SIMULATOR_THREAD_PARALLEL) && defined(_OPENMP)
104     use omp_lib
105 #endif
106     use phantom_grape_g5_x86
107     implicit none
108     integer(c_int), intent(in), value :: n_ip,n_jp
109     type(full_particle), dimension(n_ip), intent(in) :: ep_i
110     type(fdps_spj_monopole), dimension(n_jp), intent(in) :: ep_j
111     type(full_particle), dimension(n_ip), intent(inout) :: f
112     !* Local variables
113     integer(c_int) :: i,j
114     integer(c_int) :: npipe,npjpipe,devid
115     real(c_double), dimension(3,n_ip) :: xi,ai
116     real(c_double), dimension(n_ip) :: pi
117     real(c_double), dimension(3,n_jp) :: xj
118     real(c_double), dimension(n_jp) :: mj
119
120     npipe = n_ip
121     npjpipe = n_jp
122     do i=1,n_ip
123         xi(1,i) = ep_i(i)%pos%x
124         xi(2,i) = ep_i(i)%pos%y
125         xi(3,i) = ep_i(i)%pos%z
126         ai(1,i) = 0.0d0
127         ai(2,i) = 0.0d0
128         ai(3,i) = 0.0d0
129         pi(i)   = 0.0d0
130     end do
131     do j=1,n_jp
132         xj(1,j) = ep_j(j)%pos%x
133         xj(2,j) = ep_j(j)%pos%y
134         xj(3,j) = ep_j(j)%pos%z
135         mj(j)   = ep_j(j)%mass
136     end do
137 #if defined(PARTICLE_SIMULATOR_THREAD_PARALLEL) && defined(_OPENMP)
138     devid = omp_get_thread_num()
139     ! [IMPORTANT NOTE]
140     !   The subroutine calc_gravity_psp is called by a OpenMP thread
141     !   in the FDPS. This means that here is already in the parallel
142     !   region.
143     !   So, you can use omp_get_thread_num() without !$OMP parallel
144     !   directives.
145     !   If you use them, a nested parallel regions is made and the
146     !   gravity
147     !   calculation will not be performed correctly.
148 #else
149     devid = 0
150 #endif
151     call g5_set_xmjMC(devid, 0, n_jp, xj, mj)
152     call g5_set_nMC(devid, n_jp)

```

```

150      call g5_calculate_force_on_xMC(devid, xi, ai, pi, n_ip)
151      do i=1,n_ip
152          f(i)%acc%x = f(i)%acc%x + ai(1,i)
153          f(i)%acc%y = f(i)%acc%y + ai(2,i)
154          f(i)%acc%z = f(i)%acc%z + ai(3,i)
155          f(i)%pot    = f(i)%pot    - pi(i)
156      end do
157  end subroutine calc_gravity_ep_sp
158 #elseif defined(ENABLE_PIKG_KERNEL_X86)
159  subroutine calc_gravity_ep_ep(ep_i,n_ip,ep_j,n_jp,f) bind(c)
160      use, intrinsic :: iso_c_binding
161      use pikg_module_ep_ep
162      implicit none
163      integer(c_int), intent(in), value :: n_ip,n_jp
164      type(full_particle), dimension(n_ip), intent(in) :: ep_i
165      type(full_particle), dimension(n_jp), intent(in) :: ep_j
166      type(full_particle), dimension(n_ip), intent(inout) :: f
167      !* Local variables
168      integer(c_int) :: i,j
169      type(epi_grav), dimension(n_ip), target :: ep_i_tmp
170      type(epj_grav), dimension(n_jp), target :: ep_j_tmp
171      type(force_grav), dimension(n_ip), target :: f_tmp
172
173      if (n_ip > 0) then
174          do i=1,n_ip
175              ep_i_tmp(i)%pos%x = ep_i(i)%pos%x - ep_i(1)%pos%x
176              ep_i_tmp(i)%pos%y = ep_i(i)%pos%y - ep_i(1)%pos%y
177              ep_i_tmp(i)%pos%z = ep_i(i)%pos%z - ep_i(1)%pos%z
178              f_tmp(i)%acc%x = 0.0
179              f_tmp(i)%acc%y = 0.0
180              f_tmp(i)%acc%z = 0.0
181              f_tmp(i)%pot    = 0.0
182          end do
183          do j=1,n_jp
184              ep_j_tmp(j)%pos%x = ep_j(j)%pos%x - ep_i(1)%pos%x
185              ep_j_tmp(j)%pos%y = ep_j(j)%pos%y - ep_i(1)%pos%y
186              ep_j_tmp(j)%pos%z = ep_j(j)%pos%z - ep_i(1)%pos%z
187              ep_j_tmp(j)%mass = ep_j(j)%mass
188          end do
189          call pikg_calc_grav_ep_ep(c_loc(ep_i_tmp), n_ip, &
190                               c_loc(ep_j_tmp), n_jp, &
191                               c_loc(f_tmp))
192          do i=1,n_ip
193              f(i)%acc%x = f(i)%acc%x + f_tmp(i)%acc%x
194              f(i)%acc%y = f(i)%acc%y + f_tmp(i)%acc%y
195              f(i)%acc%z = f(i)%acc%z + f_tmp(i)%acc%z
196              f(i)%pot    = f(i)%pot    + f_tmp(i)%pot
197          end do
198      end if
199
200  end subroutine calc_gravity_ep_ep
201
202  subroutine calc_gravity_ep_sp(ep_i,n_ip,ep_j,n_jp,f) bind(c)
203      use, intrinsic :: iso_c_binding
204      use pikg_module_ep_ep

```

```

205 implicit none
206 integer(c_int), intent(in), value :: n_ip,n_jp
207 type(full_particle), dimension(n_ip), intent(in) :: ep_i
208 type(fdps_spj_monopole), dimension(n_jp), intent(in) :: ep_j
209 type(full_particle), dimension(n_ip), intent(inout) :: f
210 !* Local variables
211 integer(c_int) :: i,j
212 type(epi_grav), dimension(n_ip), target :: ep_i_tmp
213 type(epj_grav), dimension(n_jp), target :: ep_j_tmp
214 type(force_grav), dimension(n_ip), target :: f_tmp
215
216 if (n_ip > 0) then
217   do i=1,n_ip
218     ep_i_tmp(i)%pos%x = ep_i(i)%pos%x - ep_i(1)%pos%x
219     ep_i_tmp(i)%pos%y = ep_i(i)%pos%y - ep_i(1)%pos%y
220     ep_i_tmp(i)%pos%z = ep_i(i)%pos%z - ep_i(1)%pos%z
221     f_tmp(i)%acc%x = 0.0
222     f_tmp(i)%acc%y = 0.0
223     f_tmp(i)%acc%z = 0.0
224     f_tmp(i)%pot = 0.0
225   end do
226   do j=1,n_jp
227     ep_j_tmp(j)%pos%x = ep_j(j)%pos%x - ep_i(1)%pos%x
228     ep_j_tmp(j)%pos%y = ep_j(j)%pos%y - ep_i(1)%pos%y
229     ep_j_tmp(j)%pos%z = ep_j(j)%pos%z - ep_i(1)%pos%z
230     ep_j_tmp(j)%mass = ep_j(j)%mass
231   end do
232   call pikg_calc_grav_ep_ep(c_loc(ep_i_tmp), n_ip, &
233                             c_loc(ep_j_tmp), n_jp, &
234                             c_loc(f_tmp))
235   do i=1,n_ip
236     f(i)%acc%x = f(i)%acc%x + f_tmp(i)%acc%x
237     f(i)%acc%y = f(i)%acc%y + f_tmp(i)%acc%y
238     f(i)%acc%z = f(i)%acc%z + f_tmp(i)%acc%z
239     f(i)%pot = f(i)%pot + f_tmp(i)%pot
240   end do
241 end if
242
243 end subroutine calc_gravity_ep_sp
244 #else
245 subroutine calc_gravity_ep_ep(ep_i,n_ip,ep_j,n_jp,f) bind(c)
246   implicit none
247   integer(c_int), intent(in), value :: n_ip,n_jp
248   type(full_particle), dimension(n_ip), intent(in) :: ep_i
249   type(full_particle), dimension(n_jp), intent(in) :: ep_j
250   type(full_particle), dimension(n_ip), intent(inout) :: f
251 !* Local variables
252   integer(c_int) :: i,j
253   real(c_double) :: eps2,poti,r3_inv,r_inv
254   type(fdps_f64vec) :: xi,ai,rij
255
256 !* Compute force
257   eps2 = eps_grav * eps_grav
258   do i=1,n_ip
259     xi = ep_i(i)%pos

```

```

260      ai = 0.0d0
261      poti = 0.0d0
262      do j=1,n_jp
263          rij%x = xi%x - ep_j(j)%pos%x
264          rij%y = xi%y - ep_j(j)%pos%y
265          rij%z = xi%z - ep_j(j)%pos%z
266          r3_inv = rij%x*rij%x +
267                      + rij%y*rij%y +
268                      + rij%z*rij%z +
269                      + eps2
270          r_inv = 1.0d0/sqrt(r3_inv)
271          r3_inv = r_inv * r_inv
272          r_inv = r_inv * ep_j(j)%mass
273          r3_inv = r3_inv * r_inv
274          ai%x = ai%x - r3_inv * rij%x
275          ai%y = ai%y - r3_inv * rij%y
276          ai%z = ai%z - r3_inv * rij%z
277          poti = poti - r_inv
278          ! [IMPORTANT NOTE]
279          ! In the innermost loop, we use the components of vectors
280          ! directly for vector operations because of the following
281          ! reasion. Except for intel compilers with '-ipo' option,
282          ! most of Fortran compilers use function calls to perform
283          ! vector operations like rij = x - ep_j(j)%pos.
284          ! This significantly slow downs the speed of the code.
285          ! By using the components of vector directly, we can avoid
286          ! these function calls.
287      end do
288      f(i)%pot = f(i)%pot + poti
289      f(i)%acc = f(i)%acc + ai
290  end do
291
292 end subroutine calc_gravity_ep_ep
293
294 !**** Interaction function (particle-super particle)
295 subroutine calc_gravity_ep_sp(ep_i,n_ip,ep_j,n_jp,f) bind(c)
296     implicit none
297     integer(c_int), intent(in), value :: n_ip,n_jp
298     type(full_particle), dimension(n_ip), intent(in) :: ep_i
299     type(fdps_spj_monopole), dimension(n_jp), intent(in) :: ep_j
300     type(full_particle), dimension(n_ip), intent(inout) :: f
301     !* Local variables
302     integer(c_int) :: i,j
303     real(c_double) :: eps2,poti,r3_inv,r_inv
304     type(fdps_f64vec) :: xi,ai,rij
305
306     eps2 = eps_grav * eps_grav
307     do i=1,n_ip
308         xi = ep_i(i)%pos
309         ai = 0.0d0
310         poti = 0.0d0
311         do j=1,n_jp
312             rij%x = xi%x - ep_j(j)%pos%x
313             rij%y = xi%y - ep_j(j)%pos%y
314             rij%z = xi%z - ep_j(j)%pos%z

```

```

315      r3_inv = rij%x*rij%x &
316          + rij%y*rij%y &
317          + rij%z*rij%z &
318          + eps2
319      r_inv = 1.0d0/sqrt(r3_inv)
320      r3_inv = r_inv * r_inv
321      r_inv = r_inv * ep_j(j)%mass
322      r3_inv = r3_inv * r_inv
323      ai%x = ai%x - r3_inv * rij%x
324      ai%y = ai%y - r3_inv * rij%y
325      ai%z = ai%z - r3_inv * rij%z
326      poti = poti - r_inv
327    end do
328    f(i)%pot = f(i)%pot + poti
329    f(i)%acc = f(i)%acc + ai
330  end do
331
332  end subroutine calc_gravity_ep_sp
333 #endif
334
335 end module user_defined_types

```

Listing 35: Sample code of  $N$ -body simulation (`f_main.F90`)

```

1 !-----
2 !//////////////////////////////////////////////////////////////////////// < M A I N   R O U T I N E > /////////////////
3 !-----
4 subroutine f_main()
5   use fdps_module
6 #if defined(ENABLE_PHANTOM_GRAPE_X86)
7   use phantom_grape_g5_x86
8 #endif
9 #if defined(ENABLE_PIKG_KERNEL_X86)
10  use pikg_module_ep_ep
11 #endif
12  use user_defined_types
13  implicit none
14  !* Local parameters
15  integer, parameter :: n_tot=2**10
16  !-(force parameters)
17  real, parameter :: theta = 0.5
18  integer, parameter :: n_leaf_limit = 8
19  integer, parameter :: n_group_limit = 64
20  !-(domain decomposition)
21  real, parameter :: coef_ema=0.3
22  !-(timing parameters)
23  double precision, parameter :: time_end = 10.0d0
24  double precision, parameter :: dt = 1.0d0/128.0d0
25  double precision, parameter :: dt_diag = 1.0d0/8.0d0
26  double precision, parameter :: dt_snap = 1.0d0
27  !* Local variables
28  integer :: i,j,k,num_loop,ierr
29  integer :: psys_num,dinfo_num,tree_num
30  integer :: n_loc
31  logical :: clear
32  double precision :: ekin0,epot0,etot0

```

```

33  double precision :: ekin1,epot1,etot1
34  double precision :: time_diag,time_snap,time_sys
35  double precision :: r,acc
36  real(c_float) :: eps2
37  type(fdps_controller) :: fdps_ctrl
38  type(full_particle), dimension(:), pointer :: ptcl
39  type(c_funptr) :: pfunc_ep_ep,pfunc_ep_sp
40  !-(IO)
41  character(len=64) :: fname
42  integer(c_int) :: np
43
44  /* Initialize FDPS
45  call fdps_ctrl%PS_Initialize()
46
47  /* Create domain info object
48  call fdps_ctrl%create_dinfo(dinfo_num)
49  call fdps_ctrl%init_dinfo(dinfo_num,coef_ema)
50
51  /* Create particle system object
52  call fdps_ctrl%create_psyst(psys_num,'full_particle')
53  call fdps_ctrl%init_psyst(psys_num)
54
55  /* Make an initial condition
56  call setup_IC(fdps_ctrl,psys_num,n_tot)
57
58  /* Domain decomposition and exchange particle
59  call fdps_ctrl%decompose_domain_all(dinfo_num,psys_num)
60  call fdps_ctrl%exchange_particle(psys_num,dinfo_num)
61  n_loc = fdps_ctrl%get_nptcl_loc(psys_num)
62
63  /* Create tree object
64  call fdps_ctrl%create_tree(tree_num, &
65          "Long,full_particle,full_particle,
66          full_particle,Monopole")
67  call fdps_ctrl%init_tree(tree_num,n_loc,theta, &
68          n_leaf_limit,n_group_limit)
69
70 #if defined(ENABLE_PHANTOM_GRAPE_X86)
71   call g5_open()
72   call g5_set_eps_to_all(eps_grav);
73 #elif defined(ENABLE_PIKG_KERNEL_X86)
74   eps2 = eps_grav * eps_grav
75   call pikg_calc_grav_ep_ep_initialize(eps2)
76#endif
77
78  /* Compute force at the initial time
79  pfunc_ep_ep = c_funloc(calc_gravity_ep_ep)
80  pfunc_ep_sp = c_funloc(calc_gravity_ep_sp)
81  call fdps_ctrl%calc_force_all_and_write_back(tree_num,      &
82                                              pfunc_ep_ep, &
83                                              pfunc_ep_sp, &
84                                              psys_num,      &
85                                              dinfo_num)
86
87  /* Compute energies at the initial time
88  clear = .true.

```

```

87  call calc_energy(fdps_ctrl,psys_num,etot0,ekin0,epot0,clear)
88
89  !* Time integration
90  time_diag = 0.0d0
91  time_snap = 0.0d0
92  time_sys = 0.0d0
93  num_loop = 0
94  do
95      !* Output
96      !if (fdps_ctrl%get_rank() == 0) then
97      !    write(*,50)num_loop,time_sys
98      !    50 format('(num_loop, time_sys) = ',i5,1x,1es25.16e3)
99      !end if
100     if ( (time_sys >= time_snap) .or. &
101         (((time_sys + dt) - time_snap) > (time_snap - time_sys)) ) then
102         call output(fdps_ctrl,psys_num)
103         time_snap = time_snap + dt_snap
104     end if
105
106     !* Compute energies and output the results
107     clear = .true.
108     call calc_energy(fdps_ctrl,psys_num,etot1,ekin1,epot1,clear)
109     if (fdps_ctrl%get_rank() == 0) then
110         if ( (time_sys >= time_diag) .or. &
111             (((time_sys + dt) - time_diag) > (time_diag - time_sys)) )
112             then
113                 write(*,100)time_sys,(etot1-etot0)/etot0
114                 100 format("time: ",1es20.10e3,"energy,error: ",1es20.10e3)
115                 time_diag = time_diag + dt_diag
116             end if
117     end if
118
119     !* Leapfrog: Kick-Drift
120     call kick(fdps_ctrl,psys_num,0.5d0*dt)
121     time_sys = time_sys + dt
122     call drift(fdps_ctrl,psys_num,dt)
123
124     !* Domain decomposition & exchange particle
125     if (mod(num_loop,4) == 0) then
126         call fdps_ctrl%decompose_domain_all(dinfo_num,psys_num)
127     end if
128     call fdps_ctrl%exchange_particle(psys_num,dinfo_num)
129
130     !* Force calculation
131     pfunc_ep_ep = c_funloc(calc_gravity_ep_ep)
132     pfunc_ep_sp = c_funloc(calc_gravity_ep_sp)
133     call fdps_ctrl%calc_force_all_and_write_back(tree_num,      &
134                                               pfunc_ep_ep,  &
135                                               pfunc_ep_sp,  &
136                                               psys_num,      &
137                                               dinfo_num)
138
139     !* Leapfrog: Kick
140     call kick(fdps_ctrl,psys_num,0.5d0*dt)
141
142     !* Update num_loop

```

```

141     num_loop = num_loop + 1
142
143     !* Termination
144     if (time_sys >= time_end) then
145         exit
146     end if
147   end do
148
149 #if defined(ENABLE_PHANTOM_GRAPE_X86)
150   call g5_close()
151#endif
152
153     !* Finalize FDPS
154   call fdps_ctrl%PS_Finalize()
155
156 end subroutine f_main
157
158 !-----
159 !//////////////// S U B R O U T I N E ///////////////////
160 !//////////////// < S E T U P _ I C > ///////////////////
161 !-----
162 subroutine setup_IC(fdps_ctrl,psys_num,nptcl_glb)
163   use fdps_vector
164   use fdps_module
165   use user_defined_types
166   implicit none
167   type(fdps_controller), intent(IN) :: fdps_ctrl
168   integer, intent(IN) :: psys_num,nptcl_glb
169   !* Local parameters
170   double precision, parameter :: m_tot=1.0d0
171   double precision, parameter :: rmax=3.0d0,r2max=rmax*rmax
172   !* Local variables
173   integer :: i,j,k,ierr
174   integer :: nprocs,myrank
175   double precision :: r2,cm_mass
176   type(fdps_f64vec) :: cm_pos,cm_vel,pos
177   type(full_particle), dimension(:), pointer :: ptcl
178   character(len=64) :: fname
179
180   !* Get # of MPI processes and rank number
181   nprocs = fdps_ctrl%get_num_procs()
182   myrank = fdps_ctrl%get_rank()
183
184   !* Make an initial condition at RANK 0
185   if (myrank == 0) then
186     !* Set # of local particles
187     call fdps_ctrl%set_nptcl_loc(psys_num,nptcl_glb)
188
189     !* Create an uniform sphere of particles
190     !** get the pointer to full particle data
191     call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
192     !** initialize Mersenne twister
193     call fdps_ctrl%MT_init_genrand(0)
194     do i=1,nptcl_glb
195       ptcl(i)%id = i

```

```

196      ptcl(i)%mass = m_tot/nptcl_glb
197      do
198          ptcl(i)%pos%x = (2.0d0*fdps_ctrl%MT_genrand_res53()-1.0d0) *
199              rmax
200          ptcl(i)%pos%y = (2.0d0*fdps_ctrl%MT_genrand_res53()-1.0d0) *
201              rmax
202          ptcl(i)%pos%z = (2.0d0*fdps_ctrl%MT_genrand_res53()-1.0d0) *
203              rmax
204          r2 = ptcl(i)%pos*ptcl(i)%pos
205          if ( r2 < r2max ) exit
206      end do
207      ptcl(i)%vel = 0.0d0
208  end do
209
210      !* Correction
211      cm_pos = 0.0d0
212      cm_vel = 0.0d0
213      cm_mass = 0.0d0
214      do i=1,nptcl_glb
215          cm_pos = cm_pos + ptcl(i)%mass * ptcl(i)%pos
216          cm_vel = cm_vel + ptcl(i)%mass * ptcl(i)%vel
217          cm_mass = cm_mass + ptcl(i)%mass
218      end do
219      cm_pos = cm_pos/cm_mass
220      cm_vel = cm_vel/cm_mass
221      do i=1,nptcl_glb
222          ptcl(i)%pos = ptcl(i)%pos - cm_pos
223          ptcl(i)%vel = ptcl(i)%vel - cm_vel
224      end do
225
226      !* Output
227      !fname = 'initial.dat'
228      !open(unit=9,file=trim(fname),action='write',status='replace', &
229      !      form='unformatted',access='stream')
230      !open(unit=9,file=trim(fname),action='write',status='replace')
231      !    do i=1,nptcl_glb
232          !write(9)ptcl(i)%pos%x,ptcl(i)%pos%y,ptcl(i)%pos%z
233          !      write(9,'(3es25.16e3)')ptcl(i)%pos%x,ptcl(i)%pos%y,ptcl(i)%pos
234          !      %z
235      !    end do
236      !close(unit=9)
237
238      !* Release the pointer
239      nullify( ptcl )
240
241      !* Set the gravitational softening
242      eps_grav = 1.0d0/32.0d0
243
244  end subroutine setup_IC
245
246 !-----

```

```

247 !//////////////////////////////////////////////////////////////// S U B R O U T I N E //////////////////////////////////////////////////////////////////
248 !//////////////////////////////////////////////////////////////// < K I C K > //////////////////////////////////////////////////////////////////
249 !-----
250 subroutine kick(fdps_ctrl,psys_num,dt)
251   use fdps_vector
252   use fdps_module
253   use user_defined_types
254   implicit none
255   type(fdps_controller), intent(IN) :: fdps_ctrl
256   integer, intent(IN) :: psys_num
257   double precision, intent(IN) :: dt
258   !* Local variables
259   integer :: i,nptcl_loc
260   type(full_particle), dimension(:), pointer :: ptcl
261
262   !* Get # of local particles
263   nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
264
265   !* Get the pointer to full particle data
266   call fdps_ctrl%get_psyst_fptr(psys_num,ptcl)
267   do i=1,nptcl_loc
268     ptcl(i)%vel = ptcl(i)%vel + ptcl(i)%acc * dt
269   end do
270   nullify(ptcl)
271
272 end subroutine kick
273
274 !-----
275 !//////////////////////////////////////////////////////////////// S U B R O U T I N E //////////////////////////////////////////////////////////////////
276 !//////////////////////////////////////////////////////////////// < D R I F T > //////////////////////////////////////////////////////////////////
277 !-----
278 subroutine drift(fdps_ctrl,psys_num,dt)
279   use fdps_vector
280   use fdps_module
281   use user_defined_types
282   implicit none
283   type(fdps_controller), intent(IN) :: fdps_ctrl
284   integer, intent(IN) :: psys_num
285   double precision, intent(IN) :: dt
286   !* Local variables
287   integer :: i,nptcl_loc
288   type(full_particle), dimension(:), pointer :: ptcl
289
290   !* Get # of local particles
291   nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
292
293   !* Get the pointer to full particle data
294   call fdps_ctrl%get_psyst_fptr(psys_num,ptcl)
295   do i=1,nptcl_loc
296     ptcl(i)%pos = ptcl(i)%pos + ptcl(i)%vel * dt
297   end do
298   nullify(ptcl)
299
300 end subroutine drift
301

```

```

302 !-----  

303 !//////////////////////////////////////////////////////////////////////// S U B R O U T I N E //////////////////////////////////////////////////////////////////  

304 !//////////////////////////////////////////////////////////////////////// < C A L C _ E N E R G Y > //////////////////////////////////////////////////////////////////  

305 !-----  

306 subroutine calc_energy(fdps_ctrl,psys_num,etot,ekin,epot,clear)  

307   use fdps_vector  

308   use fdps_module  

309   use user_defined_types  

310   implicit none  

311   type(fdps_controller), intent(IN) :: fdps_ctrl  

312   integer, intent(IN) :: psys_num  

313   double precision, intent(INOUT) :: etot,ekin,epot  

314   logical, intent(IN) :: clear  

315   /* Local variables  

316   integer :: i,nptcl_loc  

317   double precision :: etot_loc,ekin_loc,epot_loc  

318   type(full_particle), dimension(:), pointer :: ptcl  

319  

320   /* Clear energies  

321   if (clear .eqv. .true.) then  

322     etot = 0.0d0  

323     ekin = 0.0d0  

324     epot = 0.0d0  

325   end if  

326  

327   /* Get # of local particles  

328   nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)  

329   call fdps_ctrl%get_psys_fptr(psys_num,ptcl)  

330  

331   /* Compute energies  

332   ekin_loc = 0.0d0  

333   epot_loc = 0.0d0  

334   do i=1,nptcl_loc  

335     ekin_loc = ekin_loc + ptcl(i)%mass * ptcl(i)%vel * ptcl(i)%vel  

336     epot_loc = epot_loc + ptcl(i)%mass * (ptcl(i)%pot + ptcl(i)%mass/  

337       eps_grav)  

338   end do  

339   ekin_loc = ekin_loc * 0.5d0  

340   epot_loc = epot_loc * 0.5d0  

341   etot_loc = ekin_loc + epot_loc  

342   call fdps_ctrl%get_sum(ekin_loc,ekin)  

343   call fdps_ctrl%get_sum(epot_loc,epot)  

344   call fdps_ctrl%get_sum(etot_loc,etot)  

345  

346   /* Release the pointer  

347   nullify(ptcl)  

348  

349 end subroutine calc_energy  

350 !-----  

351 !//////////////////////////////////////////////////////////////////////// S U B R O U T I N E //////////////////////////////////////////////////////////////////  

352 !//////////////////////////////////////////////////////////////////////// < O U T P U T > //////////////////////////////////////////////////////////////////  

353 !-----  

354 subroutine output(fdps_ctrl,psys_num)  

355   use fdps_vector

```

```

356  use fdps_module
357  use user_defined_types
358  implicit none
359  type(fdps_controller), intent(IN) :: fdps_ctrl
360  integer, intent(IN) :: psys_num
361  !* Local parameters
362  character(len=16), parameter :: root_dir="result"
363  character(len=16), parameter :: file_prefix_1st="snap"
364  character(len=16), parameter :: file_prefix_2nd="proc"
365  !* Local variables
366  integer :: i,nptcl_loc
367  integer :: myrank
368  character(len=5) :: file_num,proc_num
369  character(len=64) :: cmd,sub_dir, fname
370  type(full_particle), dimension(:), pointer :: ptcl
371  !* Static variables
372  integer, save :: snap_num=0
373
374  !* Get the rank number
375  myrank = fdps_ctrl%get_rank()
376
377  !* Get # of local particles
378  nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
379
380  !* Get the pointer to full particle data
381  call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
382
383  !* Output
384  write(file_num,"(i5.5)")snap_num
385  write(proc_num,"(i5.5)")myrank
386  fname = trim(root_dir) // "/" &
387      // trim(file_prefix_1st) // file_num // "-" &
388      // trim(file_prefix_2nd) // proc_num // ".dat"
389  open(unit=9,file=trim(fname),action='write',status='replace')
390    do i=1,nptcl_loc
391      write(9,100)ptcl(i)%id,ptcl(i)%mass, &
392                  ptcl(i)%pos%x,ptcl(i)%pos%y,ptcl(i)%pos%z, &
393                  ptcl(i)%vel%x,ptcl(i)%vel%y,ptcl(i)%vel%z
394      100 format(i8,1x,7e25.16e3)
395    end do
396  close(unit=9)
397  nullify(ptcl)
398
399  !* Update snap_num
400  snap_num = snap_num + 1
401
402 end subroutine output

```

## 5.2 SPH simulation with fixed smoothing length

In this section, we show a sample code for the SPH simulation with fixed smoothing length. This code is the same as what we described in section 4. One can create a working code by cut and paste this code and compile and link the resulted source program.

Listing 36: Sample code of SPH simulation (`user_defined.F90`)

```

1 !=====
2 ! MODULE: User defined types
3 !=====
4 module user_defined_types
5   use, intrinsic :: iso_c_binding
6   use fdps_vector
7   implicit none
8
9   !* Private parameters
10  real(kind=c_double), parameter, private :: pi=datan(1.0d0)*4.0d0
11  !* Public parameters
12  real(kind=c_double), parameter, public :: kernel_support_radius=2.5d0
13
14 !**** Force types
15 type, public, bind(c) :: force_dens !$fdps Force
16   !$fdps clear smth=keep
17   real(kind=c_double) :: dens
18   real(kind=c_double) :: smth
19 end type force_dens
20
21 type, public, bind(c) :: force_hydro !$fdps Force
22   !$fdps clear
23   type(fdps_f64vec) :: acc
24   real(kind=c_double) :: eng_dot
25   real(kind=c_double) :: dt
26 end type force_hydro
27
28 !**** Full particle type
29 type, public, bind(c) :: full_particle !$fdps FP
30   !$fdps copyFromForce force_dens (dens,dens)
31   !$fdps copyFromForce force_hydro (acc,acc) (eng_dot,eng_dot) (dt,dt)
32   real(kind=c_double) :: mass !$fdps charge
33   type(fdps_f64vec) :: pos !$fdps position
34   type(fdps_f64vec) :: vel
35   type(fdps_f64vec) :: acc
36   real(kind=c_double) :: dens
37   real(kind=c_double) :: eng
38   real(kind=c_double) :: pres
39   real(kind=c_double) :: smth !$fdps rsearch
40   real(kind=c_double) :: snds
41   real(kind=c_double) :: eng_dot
42   real(kind=c_double) :: dt
43   integer(kind=c_long_long) :: id
44   type(fdps_f64vec) :: vel_half
45   real(kind=c_double) :: eng_half
46 end type full_particle
47
48 !**** Essential particle type
49 type, public, bind(c) :: essential_particle !$fdps EPI,EPJ
50   !$fdps copyFromFP full_particle (id,id) (pos,pos) (vel,vel) (mass,
51   mass) (smth,smth) (dens,dens) (pres,pres) (snds,snds)
51   integer(kind=c_long_long) :: id !$fdps id
52   type(fdps_f64vec) :: pos !$fdps position
53   type(fdps_f64vec) :: vel

```

```

54     real(kind=c_double) :: mass !$fdps charge
55     real(kind=c_double) :: smth !$fdps rsearch
56     real(kind=c_double) :: dens
57     real(kind=c_double) :: pres
58     real(kind=c_double) :: snds
59 end type essential_particle
60
61 !* Public routines
62 public :: W
63 public :: gradW
64 public :: calc_density
65 public :: calc_hydro_force
66
67 contains
68
69 !-----
70 pure function W(dr,h)
71   implicit none
72   real(kind=c_double) :: W
73   type(fdps_f64vec), intent(in) :: dr
74   real(kind=c_double), intent(in) :: h
75 !* Local variables
76   real(kind=c_double) :: s,s1,s2
77
78   s = dsqrt(dr%x*dr%x &
79           +dr%y*dr%y &
80           +dr%z*dr%z)/h
81   s1 = 1.0d0 - s
82   if (s1 < 0.0d0) s1 = 0.0d0
83   s2 = 0.5d0 - s
84   if (s2 < 0.0d0) s2 = 0.0d0
85   W = (s1*s1*s1) - 4.0d0*(s2*s2*s2)
86   W = W * 16.0d0/(pi*h*h*h)
87
88 end function W
89
90 !-----
91 pure function gradW(dr,h)
92   implicit none
93   type(fdps_f64vec) :: gradW
94   type(fdps_f64vec), intent(in) :: dr
95   real(kind=c_double), intent(in) :: h
96 !* Local variables
97   real(kind=c_double) :: dr_abs,s,s1,s2,coef
98
99   dr_abs = dsqrt(dr%x*dr%x &
100                 +dr%y*dr%y &
101                 +dr%z*dr%z)
102   s = dr_abs/h
103   s1 = 1.0d0 - s
104   if (s1 < 0.0d0) s1 = 0.0d0
105   s2 = 0.5d0 - s
106   if (s2 < 0.0d0) s2 = 0.0d0
107   coef = - 3.0d0*(s1*s1) + 12.0d0*(s2*s2)
108   coef = coef * 16.0d0/(pi*h*h*h)

```

```

109     coef = coef / (dr_abs*h + 1.0d-6*h)
110     gradW%x = dr%x * coef
111     gradW%y = dr%y * coef
112     gradW%z = dr%z * coef
113
114 end function gradW
115
116 !**** Interaction function
117 subroutine calc_density(ep_i,n_ip,ep_j,n_jp,f) bind(c)
118     integer(kind=c_int), intent(in), value :: n_ip,n_jp
119     type(essential_particle), dimension(n_ip), intent(in) :: ep_i
120     type(essential_particle), dimension(n_jp), intent(in) :: ep_j
121     type(force_dens), dimension(n_ip), intent(inout) :: f
122     !* Local variables
123     integer(kind=c_int) :: i,j
124     type(fdps_f64vec) :: dr
125
126     do i=1,n_ip
127         f(i)%dens = 0.0d0
128         do j=1,n_jp
129             dr%x = ep_j(j)%pos%x - ep_i(i)%pos%x
130             dr%y = ep_j(j)%pos%y - ep_i(i)%pos%y
131             dr%z = ep_j(j)%pos%z - ep_i(i)%pos%z
132             f(i)%dens = f(i)%dens &
133                 + ep_j(j)%mass * W(dr,ep_i(i)%smth)
134         end do
135     end do
136
137 end subroutine calc_density
138
139 !**** Interaction function
140 subroutine calc_hydro_force(ep_i,n_ip,ep_j,n_jp,f) bind(c)
141     integer(kind=c_int), intent(in), value :: n_ip,n_jp
142     type(essential_particle), dimension(n_ip), intent(in) :: ep_i
143     type(essential_particle), dimension(n_jp), intent(in) :: ep_j
144     type(force_hydro), dimension(n_ip), intent(inout) :: f
145     !* Local parameters
146     real(kind=c_double), parameter :: C_CFL=0.3d0
147     !* Local variables
148     integer(kind=c_int) :: i,j
149     real(kind=c_double) :: mass_i,mass_j,smth_i,smth_j, &
150                         dens_i,dens_j,pres_i,pres_j, &
151                         snds_i,snds_j
152     real(kind=c_double) :: povrho2_i,povrho2_j, &
153                         v_sig_max,dr_dv,w_ij,v_sig,AV
154     type(fdps_f64vec) :: pos_i,pos_j,vel_i,vel_j, &
155                         dr,dv,gradW_ij
156
157     do i=1,n_ip
158         !* Zero-clear
159         v_sig_max = 0.0d0
160         !* Extract i-particle info.
161         pos_i = ep_i(i)%pos
162         vel_i = ep_i(i)%vel
163         mass_i = ep_i(i)%mass

```

```

164     smth_i = ep_i(i)%smth
165     dens_i = ep_i(i)%dens
166     pres_i = ep_i(i)%pres
167     snds_i = ep_i(i)%snds
168     povrho2_i = pres_i/(dens_i*dens_i)
169     do j=1,n_jp
170         !* Extract j-particle info.
171         pos_j%x = ep_j(j)%pos%x
172         pos_j%y = ep_j(j)%pos%y
173         pos_j%z = ep_j(j)%pos%z
174         vel_j%x = ep_j(j)%vel%x
175         vel_j%y = ep_j(j)%vel%y
176         vel_j%z = ep_j(j)%vel%z
177         mass_j = ep_j(j)%mass
178         smth_j = ep_j(j)%smth
179         dens_j = ep_j(j)%dens
180         pres_j = ep_j(j)%pres
181         snds_j = ep_j(j)%snds
182         povrho2_j = pres_j/(dens_j*dens_j)
183         !* Compute dr & dv
184         dr%x = pos_i%x - pos_j%x
185         dr%y = pos_i%y - pos_j%y
186         dr%z = pos_i%z - pos_j%z
187         dv%x = vel_i%x - vel_j%x
188         dv%y = vel_i%y - vel_j%y
189         dv%z = vel_i%z - vel_j%z
190         !* Compute the signal velocity
191         dr_dv = dr%x * dv%x + dr%y * dv%y + dr%z * dv%z
192         if (dr_dv < 0.0d0) then
193             w_ij = dr_dv / sqrt(dr%x * dr%x + dr%y * dr%y + dr%z * dr%z
194             )
195         else
196             w_ij = 0.0d0
197         end if
198         v_sig = snds_i + snds_j - 3.0d0 * w_ij
199         v_sig_max = max(v_sig_max, v_sig)
200         !* Compute the artificial viscosity
201         AV = - 0.5d0*v_sig*w_ij / (0.5d0*(dens_i+dens_j))
202         !* Compute the average of the gradients of kernel
203         gradW_ij = 0.5d0 * (gradW(dr,smth_i) + gradW(dr,smth_j))
204         !* Compute the acceleration and the heating rate
205         f(i)%acc%x = f(i)%acc%x - mass_j*(povrho2_i+povrho2_j+AV)*
206             gradW_ij%x
207         f(i)%acc%y = f(i)%acc%y - mass_j*(povrho2_i+povrho2_j+AV)*
208             gradW_ij%y
209         f(i)%acc%z = f(i)%acc%z - mass_j*(povrho2_i+povrho2_j+AV)*
210             gradW_ij%z
211         f(i)%eng_dot = f(i)%eng_dot &
212                         + mass_j * (povrho2_i + 0.5d0*AV) &
213                         *(dv%x * gradW_ij%x &
214                           +dv%y * gradW_ij%y &
215                           +dv%z * gradW_ij%z)
216     end do
217     f(i)%dt = C_CFL*2.0d0*smth_i/(v_sig_max*kernel_support_radius)
218   end do

```

---

```

215      ! [IMPORTANT NOTE]
216      ! In the innermost loop, we use the components of vectors
217      ! directly for vector operations because of the following
218      ! reasion. Except for intel compilers with '-ipo' option,
219      ! most of Fortran compilers use function calls to perform
220      ! vector operations like rij = x - ep_j(j)%pos.
221      ! This significantly slow downs the speed of the code.
222      ! By using the components of vector directly, we can avoid
223      ! these function calls.
224
225  end subroutine calc_hydro_force
226
227 end module user_defined_types

```

---

Listing 37: Sample code of SPH simulation (f\_main.F90)

```

1 !-----  

2 !//////////////////////////////////////////////////////////////////////// < M A I N     R O U T I N E > ///////////////////  

3 !-----  

4 subroutine f_main()  

5   use fdps_vector  

6   use fdps_module  

7   use user_defined_types  

8   implicit none  

9   !* Local parameters  

10  !-(force parameters)  

11  real, parameter :: theta = 0.5  

12  integer, parameter :: n_leaf_limit = 8  

13  integer, parameter :: n_group_limit = 64  

14  !-(domain decomposition)  

15  real, parameter :: coef_ema=0.3  

16  !-(IO)  

17  integer, parameter :: output_interval=10  

18  !* Local variables  

19  integer :: i,j,k,ierr  

20  integer :: nstep  

21  integer :: psys_num,dinfo_num  

22  integer :: tree_num_dens,tree_num_hydro  

23  integer :: n_tot,n_loc  

24  logical :: clear  

25  double precision :: time,dt,end_time  

26  type(fdps_f64vec) :: pos_ll,pos_ul  

27  type(fdps_controller) :: fdps_ctrl  

28  type(full_particle), dimension(:), pointer :: ptcl  

29  type(c_funptr) :: pfunc_ep_ep  

30  !-(IO)  

31  character(len=64) :: filename  

32  !* External routines  

33  double precision, external :: get_timestep  

34
35  !* Initialize FDPS
36  call fdps_ctrl%PS_Initialize()
37
38  !* Make an instance of ParticleSystem and initialize it
39  call fdps_ctrl%create_psyst(psys_num,'full_particle')
40  call fdps_ctrl%init_psyst(psys_num)

```

```

41
42    /* Make an initial condition and initialize the particle system
43    call setup_IC(fdps_ctrl,psys_num,end_time,pos_ll,pos_ul)
44
45    /* Make an instance of DomainInfo and initialize it
46    call fdps_ctrl%create_dinfo(dinfo_num)
47    call fdps_ctrl%init_dinfo(dinfo_num,coef_ema)
48    call fdps_ctrl%set_boundary_condition(dinfo_num,fdps_bc_periodic_xyz)
49    call fdps_ctrl%set_pos_root_domain(dinfo_num,pos_ll,pos_ul)
50
51    /* Perform domain decomposition and exchange particles
52    call fdps_ctrl%decompose_domain_all(dinfo_num,psys_num)
53    call fdps_ctrl%exchange_particle(psys_num,dinfo_num)
54
55    /* Make two tree structures
56    n_loc = fdps_ctrl%get_nptcl_loc(psys_num)
57    ** dens_tree (used for the density calculation)
58    call fdps_ctrl%create_tree(tree_num_dens, &
59                                "Short,force_dens,essential_particle,
59                                essential_particle,Gather")
60    call fdps_ctrl%init_tree(tree_num_dens,n_loc,theta, &
61                            n_leaf_limit,n_group_limit)
62
63    ** hydro_tree (used for the force calculation)
64    call fdps_ctrl%create_tree(tree_num_hydro, &
65                                "Short,force_hydro,essential_particle,
65                                essential_particle,Symmetry")
66    call fdps_ctrl%init_tree(tree_num_hydro,n_loc,theta, &
67                            n_leaf_limit,n_group_limit)
68
69    /* Compute density, pressure, acceleration due to pressure gradient
70    pfunc_ep_ep = c_funloc(calc_density)
71    call fdps_ctrl%calc_force_all_and_write_back(tree_num_dens, &
72                                                pfunc_ep_ep,      &
73                                                psys_num,        &
74                                                dinfo_num)
75    call set_pressure(fdps_ctrl,psys_num)
76    pfunc_ep_ep = c_funloc(calc_hydro_force)
77    call fdps_ctrl%calc_force_all_and_write_back(tree_num_hydro, &
78                                                pfunc_ep_ep,      &
79                                                psys_num,        &
80                                                dinfo_num)
81
82    /* Get timestep
83    dt = get_timestep(fdps_ctrl,psys_num)
84
85    /* Main loop for time integration
86    nstep = 0; time = 0.0d0
87    do
88        /* Leap frog: Initial Kick & Full Drift
89        call initial_kick(fdps_ctrl,psys_num,dt)
90        call full_drift(fdps_ctrl,psys_num,dt)
91
92        /* Adjust the positions of the SPH particles that run over
93        ! the computational boundaries.
94        call fdps_ctrl%adjust_pos_into_root_domain(psys_num,dinfo_num)

```

```

94
95      !* Leap frog: Predict
96      call predict(fdps_ctrl,psys_num,dt)
97
98      !* Perform domain decomposition and exchange particles again
99      call fdps_ctrl%decompose_domain_all(dinfo_num,psys_num)
100     call fdps_ctrl%exchange_particle(psys_num,dinfo_num)
101
102    !* Compute density, pressure, acceleration due to pressure gradient
103    pfunc_ep_ep = c_funloc(calc_density)
104    call fdps_ctrl%calc_force_all_and_write_back(tree_num_dens, &
105                                              pfunc_ep_ep,      &
106                                              psys_num,        &
107                                              dinfo_num)
108
109    call set_pressure(fdps_ctrl,psys_num)
110    pfunc_ep_ep = c_funloc(calc_hydro_force)
111    call fdps_ctrl%calc_force_all_and_write_back(tree_num_hydro, &
112                                              pfunc_ep_ep,      &
113                                              psys_num,        &
114                                              dinfo_num)
115
116    !* Get a new timestep
117    dt = get_timestep(fdps_ctrl,psys_num)
118
119    !* Leap frog: Final Kick
120    call final_kick(fdps_ctrl,psys_num,dt)
121
122    !* Output result files
123    if (mod(nstep,output_interval) == 0) then
124      call output(fdps_ctrl,psys_num,nstep)
125      call check_cnsrvd_vars(fdps_ctrl,psys_num)
126    end if
127
128    !* Output information to STDOUT
129    if (fdps_ctrl%get_rank() == 0) then
130      write(*,200)time,nstep
131      200 format("===== / &
132                  "time= ",1es25.16e3,      &
133                  "nstep= ",i6/          &
134                  "===== ")
```

```

149
150 end subroutine f_main
151
152 !-----+
153 !//SUBROUTINE S U B R O U T I N E ///////////////
154 !//SUBROUTINE < S E T U P _ I C > ///////////////
155 !-----+
156 subroutine setup_IC(fdps_ctrl,psys_num,end_time,pos_ll,pos_ul)
157   use fdps_vector
158   use fdps_module
159   use user_defined_types
160   implicit none
161   type(fdps_controller), intent(IN) :: fdps_ctrl
162   integer, intent(IN) :: psys_num
163   double precision, intent(inout) :: end_time
164   type(fdps_f64vec) :: pos_ll,pos_ul
165   !* Local variables
166   integer :: i,ii,irank
167   integer :: nprocs, myrank
168   integer :: nx_left, ny_left, nz_left
169   integer :: nx_right, ny_right, nz_right
170   integer :: nptcl_loc, nptcl_glb
171   integer :: nptcl_quot, nptcl_rem
172   integer :: i_first, i_last
173   double precision :: dens_left, dens_right
174   double precision :: eng_left, eng_right
175   double precision :: sv, mass
176   double precision :: x,y,z,dx,dy,dz
177   type(full_particle), dimension(:), pointer :: ptcl
178   character(len=5) :: proc_num
179   character(len=64) :: fname
180
181   !* Get # of MPI processes and rank number
182   nprocs = fdps_ctrl%get_num_procs()
183   myrank = fdps_ctrl%get_rank()
184
185   !* Set the box size
186   pos_ll%x = 0.0d0
187   pos_ll%y = 0.0d0
188   pos_ll%z = 0.0d0
189   pos_ul%x = 1.0d0
190   pos_ul%y = pos_ul%x / 8.0d0
191   pos_ul%z = pos_ul%x / 8.0d0
192
193   !* Set the left and right states
194   dens_left = 1.0d0
195   eng_left = 2.5d0
196   dens_right = 0.5d0
197   eng_right = 2.5d0
198
199   !* Set the separation of particle of the left state
200   dx = 1.0d0 / 128.0d0
201   dy = dx
202   dz = dx
203

```

```

204  !* Count # of particles in the computational domain
205  !** (1) Left-half
206  nx_left = 0
207  x = 0.0d0
208  do
209      nx_left = nx_left + 1
210      x = x + dx
211      if (x >= 0.5d0*pos_ul%x) exit
212  end do
213  ny_left = 0
214  y = 0.0d0
215  do
216      ny_left = ny_left + 1
217      y = y + dy
218      if (y >= pos_ul%y) exit
219  end do
220  nz_left = 0
221  z = 0.0d0
222  do
223      nz_left = nz_left + 1
224      z = z + dz
225      if (z >= pos_ul%z) exit
226  end do
227  !** (2) Right-half
228  nx_right = 0
229  x = 0.5d0*pos_ul%x
230  do
231      nx_right = nx_right + 1
232      x = x + (dens_left/dens_right)*dx
233      if (x >= pos_ul%x) exit
234  end do
235  ny_right = 0
236  y = 0.0d0
237  do
238      ny_right = ny_right + 1
239      y = y + dy
240      if (y >= pos_ul%y) exit
241  end do
242  nz_right = 0
243  z = 0.0d0
244  do
245      nz_right = nz_right + 1
246      z = z + dz
247      if (z >= pos_ul%z) exit
248  end do
249  !** (3) calculate # of particles
250  nptcl_glb = (nx_left * ny_left * nz_left) &
251      + (nx_right * ny_right * nz_right)
252  if (myrank == 0) then
253      write(*,*) 'nptcl_glb=' , nptcl_glb
254  end if
255
256  !* Set # of local particles
257  nptcl_quot = nptcl_glb / nprocs
258  nptcl_rem  = mod(nptcl_glb, nprocs)

```

```

259   if (myrank == 0) then
260     write(*,*) 'nptcl_quot=' , nptcl_quot
261     write(*,*) 'nptcl_rem=' , nptcl_rem
262   end if
263   nptcl_loc = nptcl_quot
264   if (myrank < nptcl_rem) nptcl_loc = nptcl_loc + 1
265   call fdps_ctrl%set_nptcl_loc(psyst_num, nptcl_loc)
266   i_first = 1 + nptcl_quot * myrank
267   if (myrank > (nptcl_rem-1)) then
268     i_first = i_first + nptcl_rem
269   else
270     i_first = i_first + myrank
271   end if
272   i_last = i_first + nptcl_loc - 1
273   do irank=0, nprocs-1
274     if (irank == myrank) then
275       write(*,"(4(a,1x,i8))",myrank=,myrank, &
276           ',nptcl_loc=',nptcl_loc, &
277           ',i_first=',i_first, &
278           ',i_last=',i_last
279     end if
280     call fdps_ctrl%barrier()
281   end do
282
283   !* Set local particles
284   call fdps_ctrl%get_psyst_fptr(psyst_num, ptcl)
285   sv = (pos_ul%x * pos_ul%y * pos_ul%z) / nptcl_glb ! specific volume
286   mass = 0.5d0*(dens_left+dens_right) * sv
287   i = 1
288   !** (1) Left-half
289   x = 0.0d0
290   do
291     y = 0.0d0
292     do
293       z = 0.0d0
294       do
295         if ((i_first <= i) .and. (i <= i_last)) then
296           ii = i - i_first + 1
297           ptcl(ii)%mass = mass
298           ptcl(ii)%pos%x = x
299           ptcl(ii)%pos%y = y
300           ptcl(ii)%pos%z = z
301           ptcl(ii)%dens = dens_left
302           ptcl(ii)%eng = eng_left
303           ptcl(ii)%smth = kernel_support_radius * 0.012d0
304           ptcl(ii)%id = i
305         end if
306         i = i + 1
307         z = z + dz
308         if (z >= pos_ul%z) exit
309       end do
310       y = y + dy
311       if (y >= pos_ul%y) exit
312     end do
313     x = x + dx

```

```

314      if (x >= 0.5d0*pos_ul%x) exit
315  end do
316 !** (2) Right-half
317 x = 0.5d0*pos_ul%x
318 do
319   y = 0.0d0
320   do
321     z = 0.0d0
322     do
323       if ((i_first <= i) .and. (i <= i_last)) then
324         ii = i - i_first + 1
325         ptcl(ii)%mass = mass
326         ptcl(ii)%pos%x = x
327         ptcl(ii)%pos%y = y
328         ptcl(ii)%pos%z = z
329         ptcl(ii)%dens = dens_right
330         ptcl(ii)%eng = eng_right
331         ptcl(ii)%smth = kernel_support_radius * 0.012d0
332         ptcl(ii)%id = i
333       end if
334       i = i + 1
335       z = z + dz
336       if (z >= pos_ul%z) exit
337     end do
338     y = y + dy
339     if (y >= pos_ul%y) exit
340   end do
341   x = x + (dens_left/dens_right)*dx
342   if (x >= pos_ul%x) exit
343 end do
344
345 !* Check the initial distribution
346 !write(proc_num,"(i5.5)")myrank
347 !fname = "initial" // proc_num // ".dat"
348 !open(unit=9,file=trim(fname),action='write',status='replace')
349 !  do i=1,nptcl_loc
350 !    write(9,'(3es25.16e3)')ptcl(i)%pos%x, &
351 !                                ptcl(i)%pos%y, &
352 !                                ptcl(i)%pos%z
353 !  end do
354 !close(unit=9)
355
356 !* Set the end time
357 end_time = 0.12d0
358
359 !* Inform to STDOUT
360 if (fdps_ctrl%get_rank() == 0) then
361   write(*,*)"setup..."
362 end if
363 !call fdps_ctrl%ps_finalize()
364 !stop 0
365
366 end subroutine setup_IC
367
368 !-----

```

```

369 !//////////////////////////////////////////////////////////////// S U B R O U T I N E //////////////////////////////////////////////////////////////////
370 !//////////////////////////////////////////////////////////////// < G E T _ T I M E S T E P > //////////////////////////////////////////////////////////////////
371 !-----
372 function get_timestep(fdps_ctrl,psys_num)
373   use fdps_vector
374   use fdps_module
375   use user_defined_types
376   implicit none
377   real(kind=c_double) :: get_timestep
378   type(fdps_controller), intent(in) :: fdps_ctrl
379   integer, intent(in) :: psys_num
380   /* Local variables
381   integer :: i,nptcl_loc
382   type(full_particle), dimension(:), pointer :: ptcl
383   real(kind=c_double) :: dt_loc
384
385   /* Get # of local particles
386   nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
387
388   /* Get the pointer to full particle data
389   call fdps_ctrl%get_psyst_fptr(psys_num,ptcl)
390   dt_loc = 1.0d30
391   do i=1,nptcl_loc
392     dt_loc = min(dt_loc, ptcl(i)%dt)
393   end do
394   nullify(ptcl)
395
396   /* Reduction
397   call fdps_ctrl%get_min_value(dt_loc,get_timestep)
398
399 end function get_timestep
400
401 !-----
402 !//////////////////////////////////////////////////////////////// S U B R O U T I N E //////////////////////////////////////////////////////////////////
403 !//////////////////////////////////////////////////////////////// < I N I T I A L _ K I C K > //////////////////////////////////////////////////////////////////
404 !-----
405 subroutine initial_kick(fdps_ctrl,psys_num,dt)
406   use fdps_vector
407   use fdps_module
408   use user_defined_types
409   implicit none
410   type(fdps_controller), intent(in) :: fdps_ctrl
411   integer, intent(in) :: psys_num
412   double precision, intent(in) :: dt
413   /* Local variables
414   integer :: i,nptcl_loc
415   type(full_particle), dimension(:), pointer :: ptcl
416
417   /* Get # of local particles
418   nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
419
420   /* Get the pointer to full particle data
421   call fdps_ctrl%get_psyst_fptr(psys_num,ptcl)
422   do i=1,nptcl_loc
423     ptcl(i)%vel_half = ptcl(i)%vel + 0.5d0 * dt * ptcl(i)%acc

```

```

424     ptcl(i)%eng_half = ptcl(i)%eng + 0.5d0 * dt * ptcl(i)%eng_dot
425   end do
426   nullify(ptcl)
427
428 end subroutine initial_kick
429
430 ! -----
431 !//////////////////////////////////////////////////////////////////////// S U B R O U T I N E ///////////////////////////////
432 !//////////////////////////////////////////////////////////////////////// < F U L L _ D R I F T > ///////////////////////////////
433 !-----
434 subroutine full_drift(fdps_ctrl,psys_num,dt)
435   use fdps_vector
436   use fdps_module
437   use user_defined_types
438   implicit none
439   type(fdps_controller), intent(in) :: fdps_ctrl
440   integer, intent(in) :: psys_num
441   double precision, intent(in) :: dt
442   /* Local variables
443   integer :: i,nptcl_loc
444   type(full_particle), dimension(:), pointer :: ptcl
445
446   /* Get # of local particles
447   nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
448
449   /* Get the pointer to full particle data
450   call fdps_ctrl%get_psyst_fptr(psys_num,ptcl)
451   do i=1,nptcl_loc
452     ptcl(i)%pos = ptcl(i)%pos + dt * ptcl(i)%vel_half
453   end do
454   nullify(ptcl)
455
456 end subroutine full_drift
457
458 ! -----
459 !//////////////////////////////////////////////////////////////////////// S U B R O U T I N E ///////////////////////////////
460 !//////////////////////////////////////////////////////////////////////// < P R E D I C T > ///////////////////////////////
461 !-----
462 subroutine predict(fdps_ctrl,psys_num,dt)
463   use fdps_vector
464   use fdps_module
465   use user_defined_types
466   implicit none
467   type(fdps_controller), intent(in) :: fdps_ctrl
468   integer, intent(in) :: psys_num
469   double precision, intent(in) :: dt
470   /* Local variables
471   integer :: i,nptcl_loc
472   type(full_particle), dimension(:), pointer :: ptcl
473
474   /* Get # of local particles
475   nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
476
477   /* Get the pointer to full particle data
478   call fdps_ctrl%get_psyst_fptr(psys_num,ptcl)

```

```

479 do i=1,nptcl_loc
480     ptcl(i)%vel = ptcl(i)%vel + dt * ptcl(i)%acc
481     ptcl(i)%eng = ptcl(i)%eng + dt * ptcl(i)%eng_dot
482 end do
483 nullify(ptcl)
484
485 end subroutine predict
486
487 !-----+
488 !/////////////// S U B R O U T I N E ///////////
489 !/////////////// < F I N A L _ K I C K > ///////////
490 !-----+
491 subroutine final_kick(fdps_ctrl,psys_num,dt)
492     use fdps_vector
493     use fdps_module
494     use user_defined_types
495     implicit none
496     type(fdps_controller), intent(in) :: fdps_ctrl
497     integer, intent(in) :: psys_num
498     double precision, intent(in) :: dt
499     !* Local variables
500     integer :: i,nptcl_loc
501     type(full_particle), dimension(:), pointer :: ptcl
502
503     !* Get # of local particles
504     nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
505
506     !* Get the pointer to full particle data
507     call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
508     do i=1,nptcl_loc
509         ptcl(i)%vel = ptcl(i)%vel_half + 0.5d0 * dt * ptcl(i)%acc
510         ptcl(i)%eng = ptcl(i)%eng_half + 0.5d0 * dt * ptcl(i)%eng_dot
511     end do
512     nullify(ptcl)
513
514 end subroutine final_kick
515
516 !-----+
517 !/////////////// S U B R O U T I N E ///////////
518 !/////////////// < S E T _ P R E S S U R E > ///////////
519 !-----+
520 subroutine set_pressure(fdps_ctrl,psys_num)
521     use fdps_vector
522     use fdps_module
523     use user_defined_types
524     implicit none
525     type(fdps_controller), intent(in) :: fdps_ctrl
526     integer, intent(in) :: psys_num
527     !* Local parameters
528     double precision, parameter :: hcr=1.4d0
529     !* Local variables
530     integer :: i,nptcl_loc
531     type(full_particle), dimension(:), pointer :: ptcl
532
533     !* Get # of local particles

```

```

534 nptcl_loc = fdps_ctrl%get_nptcl_loc(psyst_num)
535
536 !* Get the pointer to full particle data
537 call fdps_ctrl%get_psyst_fptr(psyst_num,ptcl)
538 do i=1,nptcl_loc
539   ptcl(i)%pres = (hcr - 1.0d0) * ptcl(i)%dens * ptcl(i)%eng
540   ptcl(i)%sns = dsqrt(hcr * ptcl(i)%pres / ptcl(i)%dens)
541 end do
542 nullify(ptcl)
543
544 end subroutine set_pressure
545
546 !-----+
547 !!!!!!! S U B R O U T I N E !!!!!!
548 !!!!!!! < O U T P U T > !!!!!!
549 !-----+
550 subroutine output(fdps_ctrl,psyst_num,nstep)
551   use fdps_vector
552   use fdps_module
553   use user_defined_types
554   implicit none
555   type(fdps_controller), intent(IN) :: fdps_ctrl
556   integer, intent(IN) :: psyst_num
557   integer, intent(IN) :: nstep
558   !* Local parameters
559   character(len=16), parameter :: root_dir="result"
560   character(len=16), parameter :: file_prefix_1st="snap"
561   character(len=16), parameter :: file_prefix_2nd="proc"
562   !* Local variables
563   integer :: i,nptcl_loc
564   integer :: myrank
565   character(len=5) :: file_num,proc_num
566   character(len=64) :: cmd,sub_dir,fname
567   type(full_particle), dimension(:), pointer :: ptcl
568
569   !* Get the rank number
570   myrank = fdps_ctrl%get_rank()
571
572   !* Get # of local particles
573   nptcl_loc = fdps_ctrl%get_nptcl_loc(psyst_num)
574
575   !* Get the pointer to full particle data
576   call fdps_ctrl%get_psyst_fptr(psyst_num,ptcl)
577
578   !* Output
579   write(file_num,"(i5.5)")nstep
580   write(proc_num,"(i5.5)")myrank
581   fname = trim(root_dir) // "/" &
582     // trim(file_prefix_1st) // file_num // "-" &
583     // trim(file_prefix_2nd) // proc_num // ".dat"
584   open(unit=9,file=trim(fname),action='write',status='replace')
585   do i=1,nptcl_loc
586     write(9,100)ptcl(i)%id,ptcl(i)%mass, &
587                 ptcl(i)%pos%x,ptcl(i)%pos%y,ptcl(i)%pos%z, &
588                 ptcl(i)%vel%x,ptcl(i)%vel%y,ptcl(i)%vel%z, &

```

```

589           ptcl(i)%dens,ptcl(i)%eng,ptcl(i)%pres
590      100 format(i8,1x,10e25.16e3)
591   end do
592 close(unit=9)
593 nullify(ptcl)
594
595 end subroutine output
596
597 !-----+
598 !////////////// S U B R O U T I N E //////////////////
599 !////////////// < C H E C K _ C N S R V D _ V A R S > //////////////////
600 !-----+
601 subroutine check_cnsrvd_vars(fdps_ctrl,psys_num)
602   use fdps_vector
603   use fdps_module
604   use user_defined_types
605   implicit none
606   type(fdps_controller), intent(in) :: fdps_ctrl
607   integer, intent(in) :: psys_num
608   !* Local variables
609   integer :: i,nptcl_loc
610   type(full_particle), dimension(:), pointer :: ptcl
611   type(fdps_f64vec) :: mom_loc,mom
612   real(kind=c_double) :: eng_loc,eng
613
614   !* Get # of local particles
615   nptcl_loc = fdps_ctrl%get_nptcl_loc(psys_num)
616
617   !* Get the pointer to full particle data
618   call fdps_ctrl%get_psys_fptr(psys_num,ptcl)
619   mom_loc = 0.0d0; eng_loc = 0.0d0
620   do i=1,nptcl_loc
621     mom_loc = mom_loc + ptcl(i)%vel * ptcl(i)%mass
622     eng_loc = eng_loc + ptcl(i)%mass *
623               *(ptcl(i)%eng +
624                 +0.5d0*ptcl(i)%vel*ptcl(i)%vel)
625   end do
626   nullify(ptcl)
627
628   !* Reduction & output
629   call fdps_ctrl%get_sum(eng_loc,eng)
630   call fdps_ctrl%get_sum(mom_loc%x,mom%x)
631   call fdps_ctrl%get_sum(mom_loc%y,mom%y)
632   call fdps_ctrl%get_sum(mom_loc%z,mom%z)
633   if (fdps_ctrl%get_rank() == 0) then
634     write(*,100)eng
635     write(*,100)mom%x
636     write(*,100)mom%y
637     write(*,100)mom%z
638     100 format(1es25.16e3)
639   end if
640
641 end subroutine check_cnsrvd_vars

```

# 6 Extensions

## 6.1 P<sup>3</sup>M code

In this section, we explain the usage of a FDPS extension “Particle Mesh” (hereafter PM) using a sample program for P<sup>3</sup>M(Particle-Particle-Particle-Mesh) method. The sample code calculates the crystal energy of sodium chloride (NaCl) crystal using the P<sup>3</sup>M method and compares the result with the analytical solution. In the P<sup>3</sup>M method, the calculation of force and potential energy is performed by splitting into Particle-Particle(PP) part and Particle-Mesh(PM) part. In this sample code, the PP part is calculated by using FDPS standard features and the PM part is computed by using a FDPS extension “PM”. Note that the detail of the extension “PM” is described in § 9.2 of the specification of FDPS and please see it for detail.

### 6.1.1 Location of sample code and working directory

The sample code is placed at \$(FDPS)/sample/fortran/p3m. Change the current directory to there.

```
$ cd $(FDPS)/sample/fortran/p3m
```

The sample code consists of `user_defined.F90` where user-defined types and interaction functions are implemented, `f_main.F90` where the other parts of the user code are implemented, and Makefiles for GCC and intel compiler, `Makefile` and `Makefile.intel`.

### 6.1.2 User-defined types

In this section, we describe derived data types that you need to define in order to perform P<sup>3</sup>M calculation using FDPS.

#### 6.1.2.1 FullParticle type

You must define a `FullParticle` type. Listing 38 shows the implementation of `FullParticle` type in the sample code. `FullParticle` type must have all physical quantities required to perform a calculation with P<sup>3</sup>M method.

Listing 38: FullParticle type

---

```

1  type, public, bind(c) :: nbody_fp !$fdps FP
2      !$fdps copyFromForce nbody_pp_results (pot,pot) (agrv,agrv)
3      !$fdps copyFromForcePM agrv_pm
4      integer(kind=c_long_long) :: id
5      real(kind=c_double) :: m !$fdps charge
6      real(kind=c_double) :: rc !$fdps rsearch
7      type(fdps_f64vec) :: x !$fdps position
8      type(fdps_f64vec) :: v,v_half
9      type(fdps_f64vec) :: agrv
10     real(kind=c_double) :: pot
11     type(fdps_f32vec) :: agrv_pm
12     real(kind=c_float) :: pot_pm

```

---

```
13    end type nbody_fp
```

At first, users must specify which user-defined type this derived data type corresponds to. The following directive specify that this derived data type is a **FullParticle** type:

```
type, public, bind(c) :: nbody_fp !$fdps FP
```

In this P<sup>3</sup>M code, the interaction force is long-range force with cutoff. Therefore, a cutoff radius is also necessary physical quantity in addition to the position and mass (charge). In the current version of FDPS, designation of cutoff radius is done by the same directive used for search radius (see § 4.2). We can tell FDPS which member variables represent these necessary quantities in the following way:

```
real(kind=c_double) :: m !$fdps charge
real(kind=c_double) :: rc !$fdps rsearch
type(fdps_f64vec) :: x !$fdps position
```

**FullParticle** type copies data from a **Force** type. Users must specify how the data is copied by using of directives. Also, when using the FDPS extension “PM” to calculate interaction, users must specify how a **FullParticle** type receives the result of interaction calculation from a “PM” module. In this sample code, there directives are written as follows.

```
!$fdps copyFromForce nbody_pp_results (pot,pot) (agrv,agrv)
!$fdps copyFromForcePM agrv_pm
```

### 6.1.2.2 EssentialParticleI type

You must define a **EssentialParticleI** type. **EssentialParticleI** type must have member variables that store all physical quantities necessary for an *i* particle to perform the PP part of the Force calculation. In the sample code, it is also used as **EssentialParticleJ** type. Therefore, it should have member variables that store all physical quantities necessary for a *j* particle to perform the PP part of the Force calculation. Listing 39 shows the implementation of **EssentialParticleI** type in the sample code.

Listing 39: EssentialParticleI 型

---

```
1  type, public, bind(c) :: nbody_ep !$fdps EPI,EPJ
2    !$fdps copyFromFP nbody_fp (id,id) (m,m) (rc,rc) (x,x)
3    integer(kind=c_long_long) :: id
4    real(kind=c_double) :: m !$fdps charge
5    real(kind=c_double) :: rc !$fdps rsearch
6    type(fdps_f64vec) :: x !$fdps position
7  end type nbody_ep
```

At first, users must tell FDPS this derived data type corresponds to **EssentialParticleI** and **EssentialParticleJ** types using a directive. This is done as follows.

```
type, public, bind(c) :: nbody_ep !$fdps EPI,EPJ
```

Next, users must specify which member variable corresponds to which necessary quantity

using a directive. As described in the explanation of `FullParticle` type, cutoff radius is also necessary quantity. Therefore, the following directives are written in this sample code.

```
real(kind=c_double) :: m !$fdps charge
real(kind=c_double) :: rc !$fdps rsearch
type(fdps_f64vec) :: x !$fdps position
```

Both `EssentialParticleI` and `EssentialParticleJ` types copy data from a `FullParticle` type. Users must specify how data copy is performed by using of directives. In this sample code, the directives are written as follows.

```
!$fdps copyFromFP nbody_fp (id,id) (m,m) (rc,rc) (x,x)
```

### 6.1.2.3 Force type

You must define a `Force` type. `Force` type must have member variables that store the results of the PP part of the Force calculation. Listing 40 shows the implementation of `Force` type in this sample code. Because we consider Coulomb interaction only, one `Force` type is defined.

Listing 40: Force 型

---

```
1 type, public, bind(c) :: nbody_pp_results !$fdps Force
2   !$fdps clear
3   real(kind=c_double) :: pot
4   type(fdps_f64vec) :: agrv
5 end type nbody_pp_results
```

At first, users must specify this derived data type is a `Force` type using a directive. In this sample code, it is written as.

```
type, public, bind(c) :: nbody_pp_results !$fdps Force
```

Because this derived data type is a `Force` type, users **must** specify how member variables are initialized before interaction calculation via directives. In this sample code, we adopt the default initialization for all of the member variables. This is realized by writing a FDPS directive with `clear` keyword only:

```
!$fdps clear
```

### 6.1.2.4 calcForceEpEp

You must define an interaction function `calcForceEpEp`. `calcForceEpEp` must contain actual code for the PP part of the Force calculation and must be implemented as subroutine . Its arguments is an array of `EssentialParticleI` objects, the number of `EssentialParticleI` objects, an array of `EssentialParticleJ` objects, the number of `EssentialParticleJ` objects, and an array of `Force` objects. Listing 41 shows the implementation of `calcForceEpEp` in this sample code.

Listing 41: Interaction function calcForceEpEp

---

```

1 subroutine calc_force_ep_ep(ep_i,n_ip,ep_j,n_jp,f) bind(c)
2 integer(c_int), intent(in), value :: n_ip,n_jp
3 type(nbody_ep), dimension(n_ip), intent(in) :: ep_i
4 type(nbody_ep), dimension(n_jp), intent(in) :: ep_j
5 type(nbody_pp_results), dimension(n_ip), intent(inout) :: f
6 !* Local variables
7 integer(c_int) :: i,j
8 real(c_double) :: rij,rinv,rinv3,xi
9 type(fdps_f64vec) :: dx
10
11 do i=1,n_ip
12     do j=1,n_jp
13         dx%x = ep_i(i)%x%x - ep_j(j)%x%x
14         dx%y = ep_i(i)%x%y - ep_j(j)%x%y
15         dx%z = ep_i(i)%x%z - ep_j(j)%x%z
16         rij = dsqrt(dx%x * dx%x &
17                      + dx%y * dx%y &
18                      + dx%z * dx%z)
19         if ((ep_i(i)%id == ep_j(j)%id) .and. (rij == 0.0d0)) cycle
20         rinv = 1.0d0/rij
21         rinv3 = rinv*rinv*rinv
22         xi = 2.0d0*rij/ep_i(i)%rc
23         f(i)%pot = f(i)%pot + ep_j(j)%m * S2_pcut(xi) * rinv
24         f(i)%agrv%x = f(i)%agrv%x + ep_j(j)%m * S2_fcut(xi) * rinv3 *
25                         dx%x
26         f(i)%agrv%y = f(i)%agrv%y + ep_j(j)%m * S2_fcut(xi) * rinv3 *
27                         dx%y
28         f(i)%agrv%z = f(i)%agrv%z + ep_j(j)%m * S2_fcut(xi) * rinv3 *
29                         dx%z
30     end do
31     !* Self-interaction term
32     f(i)%pot = f(i)%pot - ep_i(i)%m * (208.0d0/(70.0d0*ep_i(i)%rc))
33 end do
34
35 end subroutine calc_force_ep_ep

```

---

The PP part in the P<sup>3</sup>M method is a two-body interaction with cutoff (i.e. the interaction is truncated if the distance between the particles is larger than the cutoff distance). Hence, cutoff functions (`S2_pcut()`, `S2_fcut()`) appears in the calculations of potential and acceleration. These cutoff functions must be the ones that are constructed assuming that the particle shape function is  $S2(r)$ , which is introduced by Hockney & Eastwood (1988)(Eq.(8.3)) and takes the form of

$$S2(r) = \begin{cases} \frac{48}{\pi a^4} \left( \frac{a}{2} - r \right) & r < a/2, \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

where  $r$  is the distance from the center of a particle,  $a$  is the scale length of the shape function. When assuming this shape function, the charge density distribution due to a particle,  $\rho(r)$ , is expressed as  $\rho(r) = q S2(r)$ , where  $q$  is the charge of the particle. Thus,  $S2(r)$  shape function gives linear density distribution. The reason why we have to use the cutoff functions that correspond to  $S2(r)$  shape function is that the cutoff functions used in the PM part also assumes the  $S2(r)$  shape function (the cutoff functions in the PM and PP

parts should be consistent with each other).

The cutoff functions must be defined by a user. Possible implementations for `S2_pcut()` and `S2_fcut()` are given at the beginning of the sample code (see the lines 22-72 in `main.cpp`). In these examples, we used Eqs.(8-72) and (8-75) in Hockney & Eastwood (1988) and we define them such that the PP interaction takes of the form:

$$\Phi_{\text{PP}}(\mathbf{r}) = \frac{m}{|\mathbf{r} - \mathbf{r}'|} \text{S2\_pcut}(\xi) \quad (2)$$

$$\mathbf{f}_{\text{PP}}(\mathbf{r}) = \frac{m(\mathbf{r} - \mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|^3} \text{S2\_fcut}(\xi) \quad (3)$$

where  $\xi = 2|\mathbf{r} - \mathbf{r}'|/a$ . In this sample code,  $a$  is expressed as a variable `rc`.

As is clear from Eq.(8-75) in Hockney & Eastwood (1988), the mesh potential  $\phi^m$  has a finite value at  $r = 0$  (we omit a factor  $1/4\pi\varepsilon_0$  here):

$$\phi^m(0) = \frac{208}{70a} \quad (4)$$

This term is taken into account the last line in the  $i$ -particle loop:

---

```
1 f(i)%pot = f(i)%pot - ep_i(i)%m * (208.0d0/(70.0d0*ep_i(i)%rc))
```

---

Note that this term is necessary to match the numerical result with the analytical solution.

### 6.1.2.5 calcForceEpSp

You must define an interaction function `calcForceEpSp`<sup>4)</sup>. `calcForceEpSp` must contain actual code for particle-superparticle interaction and must be implemented as subroutine. Its arguments is an array of `EssentialParticleI` objects, the number of `EssentialParticleI` objects, an array of `SuperParticleJ` objects, the number of `SuperParticleJ` objects, and an array of `Force` objects. Listing 42 shows the implementation of `calcForceEpSp` in the sample code.

Listing 42: Interaction function `calcForceEpSp`

---

```
1 subroutine calc_force_ep_sp(ep_i,n_ip,ep_j,n_jp,f) bind(c)
2     integer(c_int), intent(in), value :: n_ip,n_jp
3     type(nbody_ep), dimension(n_ip), intent(in) :: ep_i
4     type(fdps_spj_monopole_cutoff), dimension(n_jp), intent(in) :: ep_j
5     type(nbody_pp_results), dimension(n_ip), intent(inout) :: f
6     !* Local variables
7     integer(c_int) :: i,j
8     real(c_double) :: rij,rinv,rinv3,xi
9     type(fdps_f64vec) :: dx
10
11    do i=1,n_ip
12        do j=1,n_jp
13            dx%x = ep_i(i)%x%x - ep_j(j)%pos%x
```

---

<sup>4)</sup>As describe at the beginning of this section, the sample code uses P<sup>3</sup>M for the calculation of interaction. In order to realize it using FDPS, we perform the calculation of interaction with the opening angle criterion  $\theta$  of 0. Hence, particle-superparticle interaction should not occur. However, API `calc_force_all_and_write_back` requires a function pointer of a subroutine that calculates particle-superparticle interaction. Therefore, we defined `calcForceEpSp` here.

---

```

14      dx%y = ep_i(i)%x%y - ep_j(j)%pos%y
15      dx%z = ep_i(i)%x%z - ep_j(j)%pos%z
16      rij = dsqrt(dx%x * dx%x &
17                      + dx%y * dx%y &
18                      + dx%z * dx%z)
19      rinv = 1.0d0/rij
20      rinv3 = rinv*rinv*rinv
21      xi = 2.0d0*rij/ep_i(i)%rc
22      f(i)%pot = f(i)%pot + ep_j(j)%mass * S2_pcut(xi) * rinv
23      f(i)%agrv%x = f(i)%agrv%x + ep_j(j)%mass * S2_fcut(xi) * rinv3
24                      * dx%x
24      f(i)%agrv%y = f(i)%agrv%y + ep_j(j)%mass * S2_fcut(xi) * rinv3
25                      * dx%y
25      f(i)%agrv%z = f(i)%agrv%z + ep_j(j)%mass * S2_fcut(xi) * rinv3
25                      * dx%z
26      end do
27  end do
28
29  end subroutine calc_force_ep_sp

```

---

### 6.1.3 Main body of the sample code

In this section, we explain the main body of the sample code. Before going into details, we first give a simple explanation about the content and the structure of the sample code. As described in § 6.1, this code computes the crystal energy of NaCl crystal using the P<sup>3</sup>M method and compares the result with the analytical solution. The NaCl crystal is expressed as an uniform grid of particles in this sample code. Na and Cl are placed in the staggered layout. Particles corresponding to Na has a positive charge, while those corresponding to Cl has a negative charge. We place a crystal expressed as an grid of charged particles into a periodic computational box of the sizes [0, 1)<sup>3</sup> and calculates the crystal energy. The computational accuracy of the crystal energy should depend on the number of particles and the configuration of particles (to the grid used in the PM calculation). Hence, in the sample code, we measure the relative energy errors for a different set of these parameters and output the result of the comparisons into a file.

The structure of the sample code is as follows:

- (1) Create and initialize FDPS objects
- (2) Create a NaCl crystal for given number of particles and configuration (in subroutine `setup_NaCl_crystal()`)
- (3) Compute the potential energy of each particle by the P<sup>3</sup>M method ( In `f_main()` )
- (4) Compute the total energy of the crystal and compare it with the analytical solution (subroutine `calc_energy_error()`)
- (5) Repeat (2)-(4)

In the following, we explain in detail each steps described above.

#### 6.1.3.1 Creation of an object of type `fdps_controller`

In the FDPS Fortran interface, all APIs of FDPS are provided as member functions in the class `FDPS_controller`. This class is defined in the module `fdps_module` in `FDPS_`

`module.F90`. Thus, in order to use APIs, the user must create an object of type `FDPS_controller`. In this sample, the object of type `FDPS_controller`, `fdps_ctrl`, is created in the main routine. Thus, in the following examples, APIs of FDPS are called as a member function of this object.

Listing 43: Creation of an object of type `fdps_controller`

---

```

1 subroutine f_main()
2   use fdps_module
3   implicit none
4   !* Local variables
5   type(fdps_controller) :: fdps_ctrl
6
7   ! Do something
8
9 end subroutine f_main

```

---

Note that the code shown above is an only necessary part from the sample code.

### 6.1.3.2 Initialization and Termination of FDPS

First, you must initialize FDPS by the following code.

Listing 44: Initialization of FDPS

---

```

1 fdps_ctrl%ps_initialize();

```

---

Once started, FDPS should be terminated explicitly. In this sample, FDPS is terminated just before the termination of the program. Hence, you need to write the following code at the end of the main function.

Listing 45: Termination of FDPS

---

```

1 fdps_ctrl%ps_finalize();

```

---

### 6.1.3.3 Creation and initialization of FDPS objects

After the initialization of FDPS, a user need to create the objects used to talk to FDPS. In this section, we describe how to create and initialize these objects.

#### 6.1.3.3.1 Creation of necessary FDPS objects

In the calculation using the P<sup>3</sup>M method, we must create `ParticleSystem` and `DomainInfo` objects. In addition, `Tree` and `ParticleMesh` objects are also needed to calculate the PP and PM parts of the force calculation.

Listing 46: Creation of FDPS objects

---

```

1 call fdps_ctrl%create_psyst(psys_num, 'nbody_fp')
2 call fdps_ctrl%create_dinfo(dinfo_num)
3 call fdps_ctrl%create_pm(pm_num)
4 call fdps_ctrl%create_tree(tree_num, &
                           "Long, nbody_pp_results, nbody_ep, nbody_fp,
                           MonopoleWithCutoff")

```

---

Note that the code snippet shown above differs from the actual sample code.

#### 6.1.3.3.2 Initialization of FDPS objects

After the creation of FDPS objects, you must initialize these objects before you use them in a user code. In the following, we explain how to initialize each object.

(i) *Initialization of a ParticleSystem object* A ParticleSystem object is initialized as follows:

Listing 47: Initialization of a ParticleSystem object

---

```
1 call fdps_ctrl%init_psys(psys_num)
```

---

This is done in the main routine in the sample code.

(ii) *Initialization of a DomainInfo object* A DomainInfo object is initialized as follows:

Listing 48: Initialization of a DomainInfo object

---

```
1 call fdps_ctrl%init_dinfo(dinfo_num, coef_ema)
```

---

This is done in the main routine in the sample code.

After the initialization, you need to specify the boundary condition and the size of the simulation box through APIs `set_boundary_condition` and `set_pos_root_domain`. In the sample code, these procedures are performed in subroutine `setup_NaCl_crystal` that sets up the distribution of particles:

---

```
1 call fdps_ctrl%set_boundary_condition(dinfo_num, fdps_bc_periodic_xyz)
2 pos_ll%x = 0.0d0; pos_ll%y = 0.0d0; pos_ll%z = 0.0d0
3 pos_ul%x = 1.0d0; pos_ul%y = 1.0d0; pos_ul%z = 1.0d0
4 call fdps_ctrl%set_pos_root_domain(dinfo_num, pos_ll, pos_ul)
```

---

(iii) *Initialization of a Tree object* A Tree object is initialized by API `init_tree`:

Listing 49: Initialization of a Tree object

---

```
1 call fdps_ctrl%init_tree(tree_num, 3*nptcl_loc, theta, &
2 n_leaf_limit, n_group_limit)
```

---

You need to give a rough number of particles to this API as the second argument. Here, we set three times the number of local particles at the time of calling. The third argument of this API is an optional argument and represents the opening angle criterion  $\theta$  for the tree method. In the sample, we do not use the tree method in the PP part of the force calculation. Therefore, we set  $\theta = 0$ .

(iv) *Initialization of a ParticleMesh object* No explicit initialization is needed.

### 6.1.3.4 Generation of a distribution of particles

In this section, we explain subroutine `setup_NaCl_crystal` that generates a distribution of particles, and FDPS APIs called within it. Given the number of particles per one space dimension and the position of the particle that is nearest to the origin  $(0, 0, 0)$ , subroutine `setup_NaCl_crystal` makes a three-dimensional uniform grid of particles. These parameters are specified through an object of derived data type `crystal_parameters`, `NaCl_params`:

---

```

1 ! In user_defined.F90
2 type, public, bind(c) :: crystal_parameters
3   integer(kind=c_int) :: nptcl_per_side
4   type(fdps_f64vec) :: pos_vertex
5 end type crystal_parameters
6 ! In f_main.F90
7 type(crystal_parameters) :: NaCl_params
8 call setup_NaCl_crystal(fdps_ctrl, &
9                      psys_num, &
10                     dinfo_num, &
11                     NaCl_params)

```

---

In the first half of subroutine `setup_NaCl_crystal`, it makes an uniform grid of particles based on the value of `NaCl_params`. In this process, we scale the particle charge  $m$  to satisfy the relation

$$\frac{2Nm^2}{R_0} = 1, \quad (5)$$

where  $N$  is the total number of molecules (the total number of atomic particles is  $2N$ ) and  $R_0$  is the distance to the nearest particle. This scaling is introduced just for convenience: The crystal energy can be written analytically as

$$E = -\frac{N\alpha m^2}{R_0}, \quad (6)$$

where  $\alpha$  is the Madelung constant and  $\alpha \approx 1.747565$  for the NaCl crystal (e.g. see [Kittel \(2004\) "Introduction to Solid State Physics"](#)). Thus, the crystal energy depends on the total number of particles. This is inconvenient when comparing the calculation result with the analytical solution. By scaling the particle charge as described above, the crystal energy becomes independent from  $N$ .

After generating a particle distribution, this function performs domain decomposition and particle exchange using FDPS APIs. In the following, we explain these APIs.

#### 6.1.3.4.1 Domain Decomposition

API `decompose_domain_all` of the `DomainInfo` object is used to perform domain decomposition based on the current distribution of particles:

Listing 50: Domain Decomposition

---

```

1 call fdps_ctrl%decompose_domain_all(dinfo_num, psys_num)

```

---

#### 6.1.3.4.2 Particle Exchange

API `exchange_particle` of the `ParticleSystem` object is used to exchange particles based on the current decomposed domains:

Listing 51: Particle Exchange

---

```
1 call fdps_ctrl%exchange_particle(psyst_num, dinfo_num)
```

---

#### 6.1.3.5 Interaction Calculation

After these procedures are completed, we must perform the interaction calculation. In the sample code, it is performed in the main routine.

Listing 52: Interaction calculation

---

```
1 !* [4] Compute force and potential with P^3M method
2 !* [4-1] Get the pointer to FP and # of local particles
3 nptcl_loc = fdps_ctrl%get_nptcl_loc(psyst_num)
4 call fdps_ctrl%get_psyst_fptr(psyst_num, ptcl)
5 !* [4-2] PP part
6 pfunc_ep_ep = c_funloc(calc_force_ep_ep)
7 pfunc_ep_sp = c_funloc(calc_force_ep_sp)
8 call fdps_ctrl%calc_force_all_and_write_back(tree_num,      &
9                                              pfunc_ep_ep,      &
10                                             pfunc_ep_sp,      &
11                                             psyst_num,        &
12                                             dinfo_num)
13 !* [4-3] PM part
14 call fdps_ctrl%calc_pm_force_all_and_write_back(pm_num,      &
15                                              psyst_num,        &
16                                              dinfo_num)
17 do i=1,nptcl_loc
18   pos32 = ptcl(i)%x
19   call fdps_ctrl%get_pm_potential(pm_num, pos32, ptcl(i)%pot_pm)
20 end do
21 !* [4-4] Compute the total acceleration and potential
22 do i=1,nptcl_loc
23   ptcl(i)%pot = ptcl(i)%pot - ptcl(i)%pot_pm
24   ptcl(i)%agrv = ptcl(i)%agrv - ptcl(i)%agrv_pm
25 end do
```

---

We use API `calc_force_all_and_write_back` for the PP part and API `calc_pm_force_all_and_write_back` for the PM part. After calculating the PM part, the total acceleration and total potential are computed. Please note that this summation is done by subtraction. The reason why we use subtraction is that the FDPS extension “PM” computes the potential energy assuming gravity. In other words, the FDPS extension “PM” treats a charge with  $m(> 0)$  creates negative potential. Hence, ***we need to invert the signs of potential energy and acceleration*** in order to use the FDPS extension “PM” for the Coulomb interaction calculation.

### 6.1.3.6 Calculation of relative energy error

The relative error of the crystal energy is computed in the function `calc_energy_error()`, where we assume that the analytical solution is  $E_0 \equiv 2E = -1.7475645946332$ , which is numerically evaluated by the PM<sup>3</sup>(Particle-Mesh Multipole Method).

### 6.1.4 Compile

Before compiling your program, you need to install the [FFTW\(Fast Fourier Transform in the West\) library](#). Then, edit the file `Makefile` in the working directory to set the PATHs of the locations of FFTW and FDPS to the variables `FFTW_LOC` and `FDPS_LOC`. After that, run `make`.

```
$ make
```

The execution file `p3m.x` will be created in the directory `work` if the compilation is succeeded.

### 6.1.5 Run

You must run your program using MPI with the number of MPI processes is equal to or greater than 2, because of the specification of FDPS extensions. Therefore, you should run the following command:

```
$ MPIRUN -np NPROC ./p3m.x
```

where “MPIRUN” represents the command to run your program using MPI such as `mpirun` or `mpiexec`, and “NPROC” is the number of MPI processes.

### 6.1.6 Check the result

After the program ended, a file that records the relative error of the crystal energy is output in the directory `work`. Figure 3 shows the dependency of the relative error on the number of particles used.

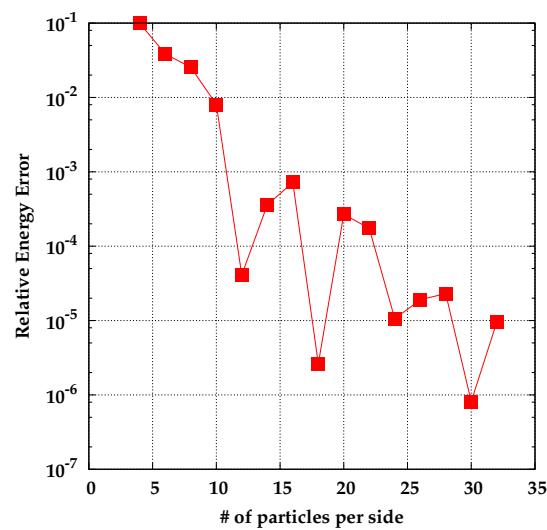


Figure 3: The relative error of the crystal energy as a function of the number of particles per side, where we assume that the number of the PM grids is  $16^3$  and the cutoff radius is  $3/16$ .

# 7 Practical Applications

In previous sections, we have explained fundamental features of FDPS using relatively simple application codes. However, we need to develop a more complex application in actual research, in which for example we need to treat different types of particles. In this section, we will explain advanced features of FDPS using practical applications. To keep the explanations short and simple, we require the readers understand the contents of the previous sections in this document.

## 7.1 *N*-body/SPH code

In this section, we explain the accompanying sample code for *N*-body/SPH simulation of a disk galaxy. In this code, dark matter and stars, which perform gravitational interaction only, are represented by *N*-body particles, while interstellar gas, which performs both gravitational and hydrodynamic interactions, is represented by SPH particles. The tree method is used for the gravity calculation. The SPH scheme adopted in this code is the one proposed by Springel & Hernquist [2002, MNRAS, 333, 649] and Springel [2005, MNRAS, 364, 1105] (hereafter, we call it Springel's SPH scheme). The readers can understand how to treat different types of particles using FDPS by reading this section.

Below, we first explain the usage of the code. Next, we give a brief explanation of the Springel's SPH scheme. Then, we explain the contents of the sample source codes in detail.

### 7.1.1 How to run the sample code

As we described, this code simulates the dynamical evolution of a disk galaxy. This code sets the initial distributions of dark matter and stars by reading a file created by MAGI (Miki & Umemura [2018, MNRAS, 475, 2269]), which is a software to make an initial condition of a galaxy simulation. On the other hand, the initial gas distribution is set inside the code. Therefore, the following procedures are required to use the code.

- Move to directory `$(FDPS)/sample/fortran/nbody+sph`
- Edit `Makefile` in the current directory
- Create particle data using MAGI and place it under directory `./magi_data/dat`
- Run the `make` command to create the executable `nbodysp.out`
- Run `nbodysp.out`
- Check the output

Below, we explain each procedure.

#### 7.1.1.1 Move to the directory the sample code is placed

Move to `$(FDPS)/sample/fortran/nbody+sph`.

### 7.1.1.2 File structure of the sample code

The following is the file structure of the sample code.

```
$ ls | awk '{print $0}'
Makefile
Makefile.K
Makefile.intel
Makefile.ofp
f_main.F90
ic.F90
job.K.sh
job.ofp.sh
leapfrog.F90
macro_defs.h
magi_data/
mathematical_constants.F90
physical_constants.F90
test.py
tipsy_file_reader.cpp
tipsy_file_reader.h
user_defined.F90
```

We explain briefly the content of each source file. In `ic.F90`, subroutines to create initial conditions are implemented. Users can choose an initial condition other than that for a disk galaxy (described later). In `leapfrog.F90`, we implement subroutines necessary to integrate the orbits of particles based on the Leapfrog method. In `macro_defs.h`, we define macros that are used to control numerical simulation. In `f_main.F90`, the main routine is implemented. In `mathematical_constants.F90`, we define some mathematical constants. In `physical_constants.F90`, we define some physical constants. In `tipsy_file_reader.*`, we define functions to read particle data created by MAGI. In `user_defined.F90`, we define user-defined types and interaction functions.

Directory `magi_data` stores a parameter file input to the software MAGI (`magi_data/cfg/*`) and a script file used to run MAGI (`magi_data/sh/run.sh`).

### 7.1.1.3 Edit Makefile

Edit `Makefile` following the description below.

- Set the variable `CXX` the command to run your C++ compiler.
- Set the variable `FC` the command to run your Fortran compiler.
- Set the variable `CXXFLAGS` compile options of the C++ compiler.
- Set the variable `FCFLAGS` compile options of the Fortran compiler.
- In this code, several macros are used to control numerical simulations. Table 1 lists the names of the macros and their definitions. In addition, there are macros whose states

(i.e. value or defined/undefined states) are automatically set according to the value of macro `INITIAL_CONDITION`. Generally, users do not have to change them. Please see `macro_defs.h` directly for detail.

- Phantom-GRAPE library for x86 can be used for the gravity calculation. To use it, set the variable `use_phantom_grape_x86 yes`.

As for the way to specify the use/non-use of OpenMP and MPI, see § 3.

Macro name	Defintion
INITIAL_CONDITION	It specifies the type of initial condition or the operation mode of the code. It must take a value from 0 to 3. According to its value, the code operates as follows. 0: an initial condition for a disk galaxy is used, 1: an initial condition for cold collapse test problem is used, 2: an initial condition for Evrard test is used, 3: the code operates in the mode to make a glass-like distribution of SPH particles.
ENABLE_VARIABLE_SMOOTHING_LENGTH	It specifies that smoothing length of SPH particles is variable or not. If it is defined, variable smoothing length is used and the SPH calculation is performed according to the Springel's SPH scheme. If it is not defined, the fixed smoothing length is used and the SPH calculation is done in almost the same way as the sample code described in § 3-4.
USE_ENTROPY	It specifies whether to use entropy or specific internal energy as an independent variable to describe the thermodynamic state of SPH particle. If defined, entropy is used. But, if macro ISOTHERMAL_EOS described below is defined, specific internal energy is forcibly used (specific internal energy is used to calculate pressure).
USE_BALSARA_SWITCH	It specifies whether Balsara switch ( <a href="#">Balsara [1995, JCP, 121, 357]</a> ) is used or not. If defined, the Balsara switch is used.
USE_PRESCR_OF_THOMAS_COUCHMAN_1992	It specifies whether a simple prescription proposed by <a href="#">Thomas &amp; Couchman [1992, MNRAS, 257, 11]</a> to prevent the tensile instability is used or not. If defined, this prescription is used.
ISOTHERMAL_EOS	It specifies whether isothermal process is assumed or not. If defined, isothermal process is assumed (specific internal energy is assumed to be constant). If not defined, the code solve the entropy equation or the internal energy equation.
READ_DATA_WITH_BYTESWAP	It specifies whether the program reads particle data with performing byte swap (byte swap is applied for each variable of basic data type). If defined, byte swap is performed.

#### 7.1.1.4 Create particle data using MAGI

As described earlier, users need to create particle data using the software MAGI before simulation according to the procedures described below. For users who cannot use MAGI for some reasons, we prepared sample particle data in web sites described below. In the following, we explain each case in detail.

**Create particle data using MAGI** Create particle data as follows.

1. Download the source file of MAGI from the web side <https://bitbucket.org/ymiki/magi> and install it in appropriate PATH according to the descriptions in Section “How to compile MAGI” in the above web side. But, our *N*-body/SPH sample code supports TIPSY file format only. Therefore, please build MAGI with `USE_TIPSY_FORMAT=ON`.
2. Edit `./magi_data/sh/run.sh` and set the variable `MAGI_INSTALL_DIR` the PATH of the directory where the `magi` command is stored. Also, set the variable `NTOT` the number of *N*-body particles (MAGI automatically assigns the numbers of dark matter particles and star particles).
3. Edit `./magi_data/cfg/*` to specify a galaxy model. For detail of the format of input file for MAGI, please see the web side above or Section 2.4 in the original paper [Miki & Umemura \[2018, MNRAS, 475, 2269\]](#). In the default, galaxy model consists of the following four components (hereafter, we call this **default galaxy model**):
  - (i) Dark matter halo (NFW profile,  $M = 10^{12} \text{ M}_\odot$ ,  $r_s = 21.5 \text{ kpc}$ ,  $r_c = 200 \text{ kpc}$ ,  $\Delta_c = 10 \text{ kpc}$ )
  - (ii) Stellar bulge (King model,  $M = 5 \times 10^{10} \text{ M}_\odot$ ,  $r_s = 0.7 \text{ kpc}$ ,  $W_0 = 5$ )
  - (iii) Thick stellar disk (Sérsic profile,  $M = 2.5 \times 10^{10} \text{ M}_\odot$ ,  $r_s = 3.5 \text{ kpc}$ ,  $n = 1.5$ ,  $z_d = 1 \text{ kpc}$ ,  $Q_{T,\min} = 1.0$ )
  - (iv) Thin stellar disk (exponential disk,  $M = 2.5 \times 10^{10} \text{ M}_\odot$ ,  $r_s = 3.5 \text{ kpc}$ ,  $z_d = 0.5 \text{ kpc}$ ,  $Q_{T,\min} = 1.0$ )

In the default galaxy model, two stellar disks are marginally unstable to a bar-mode in view of the Ostriker-Peebles criterion. Therefore, a simulated galaxy is expected to evolve into a spiral galaxy having a weak bar. In the latest release of MAGI (version 1.1.1 [as of July 19th, 2019]), its default operation mode is changed from previous releases. With this demand, we have replaced parameter  $f$  in thick and thin disks by  $Q_{T,\min}$ , where  $f$  is a parameter controlling the velocity dispersion of disk and is used in the previous releases of MAGI to specify the stability of a disk component.  $Q_{T,\min}$  is the minimum of Toomre Q value in the disk. (In the sample code in FDPS 5.0d or earlier, we used  $f = 0.125$ ).

4. Move to directory `magi_data` and run the following command:

```
$ ./sh/run.sh
```

5. If MAGI stops successfully, particle data whose extension is `tipsy` will be created in directory `magi_data/dat`.

**Download sample particle data form our web sites** Download a particle data file from one of the following URLs and place it under directory `./magi_data/dat/`. All of particle data is made with the default galaxy model. Only the number of particles is different for each data.

- $N = 2^{21}$ : [https://v2.jmlab.jp/owncloud/index.php/s/XnzvW5XAYwfqZYQ/download?path=%2Fmagi\\_data%2FGalaxy%2F21&files=Galaxy.tipsy](https://v2.jmlab.jp/owncloud/index.php/s/XnzvW5XAYwfqZYQ/download?path=%2Fmagi_data%2FGalaxy%2F21&files=Galaxy.tipsy)
- $N = 2^{22}$ : [https://v2.jmlab.jp/owncloud/index.php/s/XnzvW5XAYwfqZYQ/download?path=%2Fmagi\\_data%2FGalaxy%2F22&files=Galaxy.tipsy](https://v2.jmlab.jp/owncloud/index.php/s/XnzvW5XAYwfqZYQ/download?path=%2Fmagi_data%2FGalaxy%2F22&files=Galaxy.tipsy)
- $N = 2^{23}$ : [https://v2.jmlab.jp/owncloud/index.php/s/XnzvW5XAYwfqZYQ/download?path=%2Fmagi\\_data%2FGalaxy%2F23&files=Galaxy.tipsy](https://v2.jmlab.jp/owncloud/index.php/s/XnzvW5XAYwfqZYQ/download?path=%2Fmagi_data%2FGalaxy%2F23&files=Galaxy.tipsy)
- $N = 2^{24}$ : [https://v2.jmlab.jp/owncloud/index.php/s/XnzvW5XAYwfqZYQ/download?path=%2Fmagi\\_data%2FGalaxy%2F24&files=Galaxy.tipsy](https://v2.jmlab.jp/owncloud/index.php/s/XnzvW5XAYwfqZYQ/download?path=%2Fmagi_data%2FGalaxy%2F24&files=Galaxy.tipsy)

#### 7.1.1.5 Run make

Type “make” to run the `make` command.

#### 7.1.1.6 Run the sample code

- If you are not using MPI, run the following in CLI (terminal)

```
$ ./nbodysph.out
```

- If you are using MPI, run the following in CLI (terminal)

```
$ MPIRUN -np NPROC ./nbodysph.out
```

where `MPIRUN` should be `mpirun` or `mpiexec` depending on your MPI configuration, and `NPROC` is the number of processes you will use.

#### 7.1.1.7 Analysis of the result

In the directory `result`, data of  $N$ -body and SPH particles are output as files “`nbody0000x-proc0000y.dat`” and “`sph0000x-proc0000y.dat`”, where  $x$  is an integer representing time and  $y$  is an integer representing a process number (MPI rank number). The output file format of  $N$ -body particle data is that in each line, index of particle, mass, position ( $x$ ,  $y$ ,  $z$ ), velocity ( $v_x$ ,  $v_y$ ,  $v_z$ ) are listed. The output file format of SPH particle data is that in each line, index of particle, mass, positon ( $x$ ,  $y$ ,  $z$ ), velocity ( $v_x$ ,  $v_y$ ,  $v_z$ ), density, specific internal energy, entropy, pressure are listed.

Figure 4 shows the distribution of star and SPH particles at  $T = 0.46$  for a disk galaxy simulation with the number of  $N$ -body particles is  $2^{21}$  and the number of SPH particles is  $2^{18}$ .

Below, we briefly explain the Springel’s SPH scheme and then explain the implementation of the sample code.

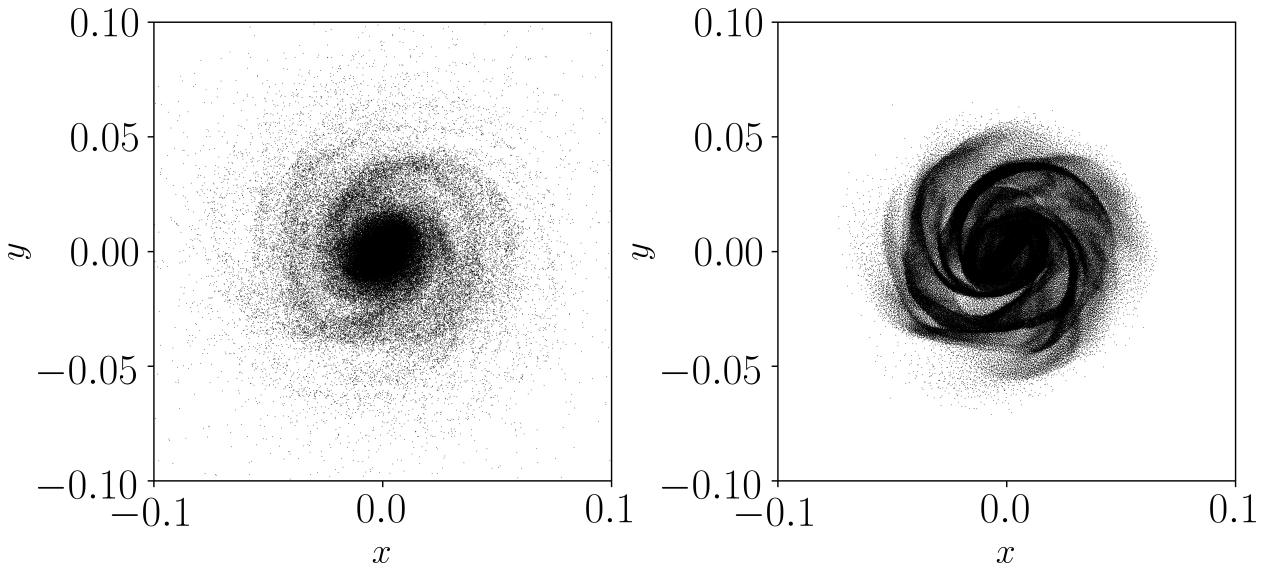


Figure 4: Face-on view of distributions of stars (left) and gas (right) (simulation configuration: the simulation is performed the number of  $N$ -body particles is  $2^{21}$ , the number of SPH particles is  $2^{18}$ , isothermal, gas temperature is  $10^4$  K, mean molecular weight to the mass of hydrogen  $\mu = 0.5$ )

### 7.1.2 Springel's SPH scheme

Springel & Hernquist [2002, MNRAS, 333, 649] proposed a formulation of SPH (actually, equation of motion[EoM]) where the total energy and entropy of a system are conserved even if smoothing length changes with time. In this section, we briefly explain their formulation. The outline of the derivation is as follows. Construct a Lagrangian of the system assuming that smoothing length is also independent variable, then solve the Euler-Lagrange equations under  $N$  constraints, where  $N$  is the number of particles.

More specifically, they consider the Lagrangian

$$L(\mathbf{q}, \dot{\mathbf{q}}) = \frac{1}{2} \sum_{i=1}^N m_i \dot{\mathbf{r}}_i^2 - \frac{1}{\gamma - 1} \sum_{i=1}^N m_i A_i \rho_i^{\gamma-1} \quad (7)$$

where  $\mathbf{q} = (\mathbf{r}_1, \dots, \mathbf{r}_N, h_1, \dots, h_N)$  is the generalized coordinates (the subscripts represent the indice of particles),  $\mathbf{r}_i$  is the position,  $h_i$  is smoothing length,  $m_i$  is mass,  $\gamma$  is the ratio of specific heats,  $\rho_i$  is density,  $A_i$  is called entropy function and it is related with specific internal energy  $u_i$  and  $\rho_i$  through the equation

$$u_i = \frac{A_i}{\gamma - 1} \rho_i^{\gamma-1} \quad (8)$$

The first and second terms of Eq.(7) represents the kinetic energy and the internal energy of the system, respectively. Because solving the Euler-Lagrange equation directly using this Lagrangian results in  $4N$  equations, which is not undesirable, they introduce the following  $N$  constraints.

$$\phi_i = \frac{4\pi}{3} h_i^3 \rho_i - \bar{m} N_{\text{neigh}} = 0 \quad (9)$$

where  $\bar{m}$  is the average mass of SPH particles<sup>5)</sup>,  $N_{\text{neigh}}$  is the number of neighbor particles (constant). Under these constraints, using the method of Lagrange multiplier, they solve the Euler-Lagrange equations to obtain the following equations of motion:

$$\frac{d\mathbf{v}_i}{dt} = - \sum_{j=1}^N m_j \left[ f_i \frac{P_i}{\rho_i^2} \nabla_i W(r_{ij}, h_i) + f_j \frac{P_j}{\rho_j^2} \nabla_i W(r_{ij}, h_j) \right] \quad (10)$$

where  $P_i$  is pressure,  $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ ,  $W$  is the kernel function,  $f_i$  is the so-called  $\nabla h$  term, defined by

$$f_i = \left( 1 + \frac{h_i}{3\rho_i} \frac{\partial \rho_i}{\partial h_i} \right)^{-1} \quad (11)$$

The thermodynamic state of the system is described by the independent variable  $A_i$ , the entropy. If the flow is adiabatic, the entropy is constant along the flow except for locations of shock waves where the entropy is increased. Springel [2005, MNRAS, 364, 1105] modeled the increase of the entropy by passing shock waves using the method of artificial viscosity:

$$\frac{dA_i}{dt} = \frac{1}{2} \frac{\gamma - 1}{\rho_i^{\gamma-1}} \sum_{j=1}^N m_j \Pi_{ij} \mathbf{v}_{ij} \cdot \nabla_i \bar{W}_{ij} \quad (12)$$

$$\frac{d\mathbf{v}_i}{dt} \Big|_{\text{visc}} = - \sum_{j=1}^N m_j \Pi_{ij} \nabla_i \bar{W}_{ij} \quad (13)$$

where  $\mathbf{v}_{ij} = \mathbf{v}_i - \mathbf{v}_j$ ,  $\mathbf{v}_i$  is velocity,  $\bar{W}_{ij} = \frac{1}{2}(W(r_{ij}, h_i) + W(r_{ij}, h_j))$ . For  $\Pi_{ij}$ , please see the original papers.

The procedures of SPH calculation is summarized as follows:

- (1) Solve Eq.(9) and the following equation self-consistently to determine the density  $\rho_i$  and the smoothing length  $h_i$ .

$$\rho_i = \sum_{j=1}^N m_j W(r_{ij}, h_i) \quad (14)$$

- (2) Calculate  $\nabla h$  term defined by Eq.(11).
- (3) Calculate the right-hand side of Eqs.(10), (12)、(13).
- (4) Update the positions, velocities, entropies of SPH particles.

In the remaining sections, we first explain the implementations of user-defined classes and interaction functions. Then, we explain the implementation of the main routine where we explain how to treat different types of particles in FDPS.

### 7.1.3 User-defined types

All user-defined types are defined in `user_defined.F90`. Here, we explain the types of user-defined types used in this code. As described earlier, this code use two types of

<sup>5)</sup>This must be treated as constant.

particles,  $N$ -body and SPH particles. Thus, this code defines **two** `FullParticle` types (`fp_nbody` type for  $N$ -body particles and `fp_sph` type for SPH particles). The number of types of *physical* interactions are two, the gravitational and hydrodynamic interactions. But, as explained in § 4, we need to perform (at least) two interaction calculations (for density and acceleration) in SPH calculations. Therefore, the code defines **three** `Force` types (`force_grav` type for the gravity calculation, `force_dens` type for the density calculation, and `force_hydro` type for the calculation of acceleration due to pressure gradient (hereafter we call it pressure-gradient acceleration for simplicity)). For simplicity, this code uses one derived data type for both `EssentialParticleI` type and `EssentialParticleJ` type (hereafter, we call them together `EssentialParticle` type). Also this code uses the same `EssentialParticle` type for the calculations of density and pressure-gradient acceleration. Therefore, the number of types of `EssentialParticle` types is **two** (`ep_grav` type for the gravity calculation and `ep_hydro` type for SPH calculation).

Below, we explain the implementation of each user defined type.

#### 7.1.3.1 FullParticle type

First, we explain derived data type `fp_nbody`, which is used to store the information of  $N$ -body particles. This data type contains all physical quantities that a  $N$ -body particle should have as member variables. Listing 53 shows the implementation of `fp_nbody` type. The definitions of the member variables are almost the same as those of  $N$ -body sample code introduced in § 3-4. Thus, please see the corresponding section for detail.

Listing 53: `FullParticle` type (`fp_nbody` type)

---

```

1 !**** Full particle type
2 type, public, bind(c) :: fp_nbody !$fdps FP
3   !$fdps copyFromForce force_grav (acc,acc) (pot,pot)
4   integer(kind=c_long_long) :: id !$fdps id
5   real(kind=c_double) :: mass !$fdps charge
6   type(fdps_f64vec) :: pos !$fdps position
7   type(fdps_f64vec) :: vel
8   type(fdps_f64vec) :: acc
9   real(kind=c_double) :: pot
10 end type fp_nbody

```

---

Next, we explain derived data type `fp_sph`, which is used to store the information of SPH particles. This data type contains all physical quantities that a SPH particle should have as member variables. Listing 54 shows the implementation of `fp_sph` type. The definitions of main member variables are as follows: `id` (identification number), `mass` (mass), `pos` (position $[r_i]$ ), `vel` (velocity $[v_i]$ ), `acc_grav` (gravitational acceleration), `pot_grav` (gravitational potential), `acc_hydro` (pressure-gradient acceleration), `dens` (density $[\rho_i]$ ), `eng` (specific internal energy $[u_i]$ ), `ent` (entropy function [hereafter, entropy] $[A_i]$ ), `pres` (pressure $[P_i]$ ), `smth` (smoothing length<sup>6)</sup> $[h_i]$ ), `gradh` ( $\nabla h$  term $[f_i]$ ), `divv` ( $(\nabla \cdot v)_i$ , where the subscript  $i$  means that the derivative is performed at particle position), `rotv` ( $(\nabla \times v)_i$ ), `balsw` (coefficient for Balsara switch and its definition is the same as  $f(a)$  in Balsara [1995, JCP, 121, 357]),

---

<sup>6)</sup>It is defined as the distance from the center of a particle where the value of the SPH kernel function is 0.

`snds` (sound speed), `eng_dot` (time rate of change of `eng`), `ent_dot` (time rate of change of `ent`), `dt` (the maximum allowable time step to integrate the orbit of this particle).

The following points should be noted.

- SPH particles are involved with three types of interaction calculations (gravity, density, pressure-gradient acceleration). Thus, **three** types of `copyFromForce` directives are written.

Listing 54: FullParticle type (fp\_sph type)

---

```

1  type, public, bind(c) :: fp_sph !$fdps FP
2    !$fdps copyFromForce force_grav (acc,acc_grav) (pot,pot_grav)
3    !$fdps copyFromForce force_dens (flag,flag) (dens,dens) (smth,smth)
4      (gradh,gradh) (divv,divv) (rotv,rotv)
5    !$fdps copyFromForce force_hydro (acc,acc_hydro) (eng_dot,eng_dot) (
6      ent_dot,ent_dot) (dt,dt)
7    integer(kind=c_long_long) :: id !$fdps id
8    real(kind=c_double) :: mass !$fdps charge
9    type(fdps_f64vec) :: pos !$fdps position
10   type(fdps_f64vec) :: vel
11   type(fdps_f64vec) :: acc_grav
12   real(kind=c_double) :: pot_grav
13   type(fdps_f64vec) :: acc_hydro
14   integer(kind=c_int) :: flag
15   real(kind=c_double) :: dens
16   real(kind=c_double) :: eng
17   real(kind=c_double) :: ent
18   real(kind=c_double) :: pres
19   real(kind=c_double) :: smth
20   real(kind=c_double) :: gradh
21   real(kind=c_double) :: divv
22   type(fdps_f64vec) :: rotv
23   real(kind=c_double) :: balsw
24   real(kind=c_double) :: snds
25   real(kind=c_double) :: eng_dot
26   real(kind=c_double) :: ent_dot
27   real(kind=c_double) :: dt
28   type(fdps_f64vec) :: vel_half
29   real(kind=c_double) :: eng_half
30   real(kind=c_double) :: ent_half
29 end type fp_sph

```

---

### 7.1.3.2 EssentialParticle type

First, we explain derived data type `ep_grav`, which is used for the gravity calculation. This data type has all physical quantities that  $i$ - and  $j$ -particles should have in order to perform gravity calculation as member variables. Listing 55 shows the implementation of `ep_grav` type. `EssentialParticle` type should have `copyFromFP` directive(s) to specify the way of copy data from `FullParticle` type(s). In this code, there are two `FullParticle` types and hence **two** `copyFromFP` directives are written.

Listing 55: EssentialParticle type (ep\_grav type)

---

```

1 type, public, bind(c) :: ep_grav !$fdps EPI,EPJ
2   !$fdps copyFromFP fp_nbody (id,id) (mass,mass) (pos,pos)
3   !$fdps copyFromFP fp_sph (id,id) (mass,mass) (pos,pos)
4   integer(kind=c_long_long) :: id !$fdps id
5   real(kind=c_double) :: mass !$fdps charge
6   type(fdps_f64vec) :: pos !$fdps position
7 end type ep_grav

```

---

Next, we explain derived data type `ep_hydro`, which is used for the calculations of density and pressure-gradient acceleration. This data type has all physical quantities that  $i$ - and  $j$ -partiles should have in order to perform the calculations of density and pressure-gradient acceleration. Listing 56 shows the implementation of `ep_hydro` type .

Listing 56: EssentialParticle type (`ep_hydro` type)

---

```

1 type, public, bind(c) :: ep_hydro !$fdps EPI,EPJ
2   !$fdps copyFromFP fp_sph (id,id) (pos,pos) (vel,vel) (mass,mass) (
3     smth,smth) (dens,dens) (pres,pres) (gradh,gradh) (sns,snds) (
4     balsw,balsw)
5   integer(kind=c_long_long) :: id !$fdps id
6   type(fdps_f64vec) :: pos !$fdps position
7   type(fdps_f64vec) :: vel
8   real(kind=c_double) :: mass !$fdps charge
9   real(kind=c_double) :: smth !$fdps rsearch
10  real(kind=c_double) :: dens
11  real(kind=c_double) :: pres
12  real(kind=c_double) :: gradh
13  real(kind=c_double) :: sns
14  real(kind=c_double) :: balsw
15 end type ep_hydro

```

---

### 7.1.3.3 Force type

First, we explain derived data type `force_grav`, which is a `Force` type used for the gravity calculation. This data type must have all physical quantities that are obtained as the result of the gravity calculation. Listing 57 shows the implementation of `force_grav` type .

Listing 57: Force type (`force_grav` type)

---

```

1 type, public, bind(c) :: force_grav !$fdps Force
2   !$fdps clear
3   type(fdps_f64vec) :: acc
4   real(kind=c_double) :: pot
5 end type force_grav

```

---

Next, we explain derived data type `force_dens`, which is a `Force` type used for the density calculation. This data type must have all physical quantities that are obtained as the result of the density calculation. Listing 58 shows the implementation of `force_dens` type . In the Springel's SPH scheme, the smoothing length  $h_i$  changes depending on the density at the position of a particle,  $\rho_i$ . In other words,  $h_i$  is also updated with  $\rho_i$ . Therefore, there is member variable `smth` to store updated smoothing length. In this code, we calculate  $\nabla h$  term,  $(\nabla \cdot \mathbf{v})_i$   $(\nabla \times \mathbf{v})_i$  at the same time (if `USE_BALSARA_SWITCH` is defined). Thus,

there are member variables `gradh`, `divv`, `rotv` to store them. Member variable `flag` is used to store the result of iteration calculation of  $\rho_i$  and  $h_i$  (for detail, see § 7.1.4.2).

Listing 58: Force type (`force_dens` type)

---

```

1  type, public, bind(c) :: force_dens !$fdps Force
2    !$fdps clear smth=keep
3    integer(kind=c_int) :: flag
4    real(kind=c_double) :: dens
5    real(kind=c_double) :: smth
6    real(kind=c_double) :: gradh
7    real(kind=c_double) :: divv
8    type(fdps_f64vec) :: rotv
9  end type force_dens

```

---

Finally, we explain derived data type `force_hydro`, which is a `Force` type used for the calculation of pressure-gradient acceleration. This data type must have all physical quantities that are obtained as the result of the calculation of pressure-gradient acceleration. Listing 59 shows the implementation of `force_hydro` type .

Listing 59: Force type (`force_hydro` type)

---

```

1  type, public, bind(c) :: force_hydro !$fdps Force
2    !$fdps clear
3    type(fdps_f64vec) :: acc
4    real(kind=c_double) :: eng_dot
5    real(kind=c_double) :: ent_dot
6    real(kind=c_double) :: dt
7  end type force_hydro

```

---

## 7.1.4 Interaction functions

All interaction functions are implemented in `user_defined.F90`. There are **three** types of interaction functions. Below, we explain them.

### 7.1.4.1 Interaction function for the gravity calculation

Interaction functions for the gravity calculation are implemented as subroutines `calc_gravity_ep_ep` and `calc_gravity_ep_sp`. Listing 60 shows the implementation. The implementation is almost the same as that of the  $N$ -body sample code introduced in § 3-4. For detail, please the corresponding section.

Listing 60: Interaction function for the gravity calculation

---

```

1 #if defined(ENABLE_PHANTOM_GRAPE_X86)
2   subroutine calc_gravity_ep_ep(ep_i,n_ip,ep_j,n_jp,f) bind(c)
3 #if defined(PARTICLE_SIMULATOR_THREAD_PARALLEL) && defined(_OPENMP)
4   use omp_lib
5 #endif
6   use phantom_grape_g5_x86
7   implicit none
8   integer(c_int), intent(in), value :: n_ip,n_jp
9   type(ep_grav), dimension(n_ip), intent(in) :: ep_i
10  type(ep_grav), dimension(n_jp), intent(in) :: ep_j

```

---

```

11      type(force_grav), dimension(n_ip), intent(inout) :: f
12      !* Local variables
13      integer(c_int) :: i,j
14      integer(c_int) :: npipe,njpipe,devid
15      real(c_double), dimension(3,n_ip) :: xi,ai
16      real(c_double), dimension(n_ip) :: pi
17      real(c_double), dimension(3,n_jp) :: xj
18      real(c_double), dimension(n_jp) :: mj
19
20      npipe = n_ip
21      njpipe = n_jp
22      do i=1,n_ip
23          xi(1,i) = ep_i(i)%pos%x
24          xi(2,i) = ep_i(i)%pos%y
25          xi(3,i) = ep_i(i)%pos%z
26          ai(1,i) = 0.0d0
27          ai(2,i) = 0.0d0
28          ai(3,i) = 0.0d0
29          pi(i)   = 0.0d0
30      end do
31      do j=1,n_jp
32          xj(1,j) = ep_j(j)%pos%x
33          xj(2,j) = ep_j(j)%pos%y
34          xj(3,j) = ep_j(j)%pos%z
35          mj(j)   = ep_j(j)%mass
36      end do
37 #if defined(PARTICLE_SIMULATOR_THREAD_PARALLEL) && defined(_OPENMP)
38     devid = omp_get_thread_num()
39     ! [IMPORTANT NOTE]
40     !   The subroutine calc_gravity_ep_ep is called by a OpenMP thread
41     !   in the FDPS. This means that here is already in the parallel
42     !   region.
43     !   So, you can use omp_get_thread_num() without !$OMP parallel
44     !   directives.
45     !   If you use them, a nested parallel regions is made and the
46     !   gravity
47     !   calculation will not be performed correctly.
48 #else
49     devid = 0
50 #endif
51     call g5_set_xmjMC(devid, 0, n_jp, xj, mj)
52     call g5_set_nMC(devid, n_jp)
53     call g5_calculate_force_on_xMC(devid, xi, ai, pi, n_ip)
54     do i=1,n_ip
55         f(i)%acc%x = f(i)%acc%x + ai(1,i)
56         f(i)%acc%y = f(i)%acc%y + ai(2,i)
57         f(i)%acc%z = f(i)%acc%z + ai(3,i)
58         f(i)%pot   = f(i)%pot   - pi(i)
59     end do
60 end subroutine calc_gravity_ep_ep
61
62 subroutine calc_gravity_ep_sp(ep_i,n_ip,ep_j,n_jp,f) bind(c)
63 #if defined(PARTICLE_SIMULATOR_THREAD_PARALLEL) && defined(_OPENMP)
64     use omp_lib
65 #endif

```

```

63      use phantom_grape_g5_x86
64      implicit none
65      integer(c_int), intent(in), value :: n_ip,n_jp
66      type(ep_grav), dimension(n_ip), intent(in) :: ep_i
67      type(fdps_spj_monopole), dimension(n_jp), intent(in) :: ep_j
68      type(force_grav), dimension(n_ip), intent(inout) :: f
69      !* Local variables
70      integer(c_int) :: i,j
71      integer(c_int) :: nipipe,njpipe,devid
72      real(c_double), dimension(3,n_ip) :: xi,ai
73      real(c_double), dimension(n_ip) :: pi
74      real(c_double), dimension(3,n_jp) :: xj
75      real(c_double), dimension(n_jp) :: mj
76
77      nipipe = n_ip
78      njpipe = n_jp
79      do i=1,n_ip
80          xi(1,i) = ep_i(i)%pos%x
81          xi(2,i) = ep_i(i)%pos%y
82          xi(3,i) = ep_i(i)%pos%z
83          ai(1,i) = 0.0d0
84          ai(2,i) = 0.0d0
85          ai(3,i) = 0.0d0
86          pi(i)   = 0.0d0
87      end do
88      do j=1,n_jp
89          xj(1,j) = ep_j(j)%pos%x
90          xj(2,j) = ep_j(j)%pos%y
91          xj(3,j) = ep_j(j)%pos%z
92          mj(j)   = ep_j(j)%mass
93      end do
94 #if defined(PARTICLE_SIMULATOR_THREAD_PARALLEL) && defined(_OPENMP)
95     devid = omp_get_thread_num()
96     ! [IMPORTANT NOTE]
97     !   The subroutine calc_gravity_ep_sp is called by a OpenMP thread
98     !   in the FDPS. This means that here is already in the parallel
99     !   region.
100    !   So, you can use omp_get_thread_num() without !$OMP parallel
101    !   directives.
102    !   If you use them, a nested parallel regions is made and the
103    !   gravity
104    !   calculation will not be performed correctly.
105 #else
106     devid = 0
107 #endif
108     call g5_set_xmjMC(devid, 0, n_jp, xj, mj)
109     call g5_set_nMC(devid, n_jp)
110     call g5_calculate_force_on_xMC(devid, xi, ai, pi, n_ip)
111     do i=1,n_ip
112         f(i)%acc%x = f(i)%acc%x + ai(1,i)
113         f(i)%acc%y = f(i)%acc%y + ai(2,i)
114         f(i)%acc%z = f(i)%acc%z + ai(3,i)
115         f(i)%pot   = f(i)%pot   - pi(i)
116     end do
117 end subroutine calc_gravity_ep_sp

```

```

115 #else
116   subroutine calc_gravity_ep_ep(ep_i,n_ip,ep_j,n_jp,f) bind(c)
117     integer(kind=c_int), intent(in), value :: n_ip,n_jp
118     type(ep_grav), dimension(n_ip), intent(in) :: ep_i
119     type(ep_grav), dimension(n_jp), intent(in) :: ep_j
120     type(force_grav), dimension(n_ip), intent(inout) :: f
121   !* Local variables
122   integer(kind=c_int) :: i,j
123   real(kind=c_double) :: eps2,poti,r3_inv,r_inv
124   type(fdps_f64vec) :: xi,ai,rij
125   !* Compute force
126   eps2 = eps_grav * eps_grav
127   do i=1,n_ip
128     xi%x = ep_i(i)%pos%x
129     xi%y = ep_i(i)%pos%y
130     xi%z = ep_i(i)%pos%z
131     ai%x = 0.0d0
132     ai%y = 0.0d0
133     ai%z = 0.0d0
134     poti = 0.0d0
135     do j=1,n_jp
136       rij%x = xi%x - ep_j(j)%pos%x
137       rij%y = xi%y - ep_j(j)%pos%y
138       rij%z = xi%z - ep_j(j)%pos%z
139       r3_inv = rij%x*rij%x +
140                 + rij%y*rij%y +
141                 + rij%z*rij%z +
142                 + eps2
143       r_inv = 1.0d0/dsqrt(r3_inv)
144       r3_inv = r_inv * r_inv
145       r_inv = r_inv * ep_j(j)%mass
146       r3_inv = r3_inv * r_inv
147       ai%x = ai%x - r3_inv * rij%x
148       ai%y = ai%y - r3_inv * rij%y
149       ai%z = ai%z - r3_inv * rij%z
150       poti = poti - r_inv
151     end do
152     f(i)%acc%x = f(i)%acc%x + ai%x
153     f(i)%acc%y = f(i)%acc%y + ai%y
154     f(i)%acc%z = f(i)%acc%z + ai%z
155     f(i)%pot = f(i)%pot + poti
156   end do
157 end subroutine calc_gravity_ep_ep
158
159 subroutine calc_gravity_ep_sp(ep_i,n_ip,ep_j,n_jp,f) bind(c)
160   integer(kind=c_int), intent(in), value :: n_ip,n_jp
161   type(ep_grav), dimension(n_ip), intent(in) :: ep_i
162   type(fdps_spj_monopole), dimension(n_jp), intent(in) :: ep_j
163   type(force_grav), dimension(n_ip), intent(inout) :: f
164   !* Local variables
165   integer(kind=c_int) :: i,j
166   real(kind=c_double) :: eps2,poti,r3_inv,r_inv
167   type(fdps_f64vec) :: xi,ai,rij
168   !* Compute force
169   eps2 = eps_grav * eps_grav

```

```

170      do i=1,n_ip
171        xi%x = ep_i(i)%pos%x
172        xi%y = ep_i(i)%pos%y
173        xi%z = ep_i(i)%pos%z
174        ai%x = 0.0d0
175        ai%y = 0.0d0
176        ai%z = 0.0d0
177        poti = 0.0d0
178        do j=1,n_jp
179          rij%x = xi%x - ep_j(j)%pos%x
180          rij%y = xi%y - ep_j(j)%pos%y
181          rij%z = xi%z - ep_j(j)%pos%z
182          r3_inv = rij%x*rij%x +
183            + rij%y*rij%y +
184            + rij%z*rij%z +
185            + eps2
186          r_inv = 1.0d0/dsqrt(r3_inv)
187          r3_inv = r_inv * r_inv
188          r_inv = r_inv * ep_j(j)%mass
189          r3_inv = r3_inv * r_inv
190          ai%x = ai%x - r3_inv * rij%x
191          ai%y = ai%y - r3_inv * rij%y
192          ai%z = ai%z - r3_inv * rij%z
193          poti = poti - r_inv
194        end do
195        f(i)%acc%x = f(i)%acc%x + ai%x
196        f(i)%acc%y = f(i)%acc%y + ai%y
197        f(i)%acc%z = f(i)%acc%z + ai%z
198        f(i)%pot = f(i)%pot + poti
199      end do
200    end subroutine calc_gravity_ep_sp
201 #endif

```

---

#### 7.1.4.2 Interaction function for the density calculation

Interaction function for the density calculation is implemented as subroutine `calc_density`. Listing 61 shows its implementation. The implementation actually used differs depending on the state of macro `ENABLE_VARIABLE_SMOOTHING_LENGTH`. If this macro is not defined, an implementation for fixed smoothing length is used. Its source code is almost the same as the interaction function for the density calculation of the SPH sample code described in § 3-4. Thus, we omit explanation for this case. Below, we explain an implementation used for the case that the above macro is defined.

As described in § 7.1.2, we need to determine the density  $\rho_i$  and smoothing length  $h_i$  at the same time by solving Eqs.(14) and (9) self-consistently. For this, we need to perform an iterative calculation. This calculation is performed in the infinite `do-enddo` loop in the code. As you'll see by reading the source code of subroutine `calc_density_wrapper` in `f_main.F90`, this sample code performs the density calculation after multiplying the smoothing lengths of all particles by a constant `SCF_smth` in order to make the density calculation efficiently. By this, we can change  $h_i$  between 0 and  $h_{\max,\text{alw}} \equiv \text{SCF\_smth} \times h_{i,0}$ , during the iteration, where  $h_{i,0}$  is the value of the smoothing length of particle  $i$  before we multiply by `SCF_smth`. This is because all of particles that is eligible to be  $j$ -particles are contained in the

current  $j$ -particle list (`ep_j`). If the iteration does not converge for some particle  $i$ , we cannot determine  $\rho_i$  and  $h_i$  for this particle by using the current  $j$  particle list because the value of the smoothing length we want to obtain will be larger than  $h_{\max,\text{alw}}$ . In this case, we need to perform the density calculation again after increasing  $h_{i,0}$ . This “outer” iteration is performed in subroutine `calc_density_wrapper` in `f_main.F90`. We will describe this subroutine in § 7.1.5.

After the infinite `do-enddo` loop, this subroutine performs the calculations of  $\nabla h$ ,  $(\nabla \cdot \mathbf{v})_i$ , and  $(\nabla \times \mathbf{v})_i$ .

Listing 61: Interaction function for the density calculation

---

```

1  subroutine calc_density(ep_i,n_ip,ep_j,n_jp,f) bind(c)
2    integer(kind=c_int), intent(in), value :: n_ip,n_jp
3    type(ep_hydro), dimension(n_ip), intent(in) :: ep_i
4    type(ep_hydro), dimension(n_jp), intent(in) :: ep_j
5    type(force_dens), dimension(n_ip), intent(inout) :: f
6    !* Local parameters
7    real(kind=c_double), parameter :: eps=1.0d-6
8    !* Local variables
9    integer(kind=c_int) :: i,j
10   integer(kind=c_int) :: n_unchanged
11   real(kind=c_double) :: M,M_trgt
12   real(kind=c_double) :: dens,drho_dh
13   real(kind=c_double) :: h,h_max_alw,h_L,h_U,dh,dh_prev
14   type(fdps_f64vec) :: dr,dv,gradW_i
15
16 #if defined(ENABLE_VARIABLE_SMOOTHING_LENGTH)
17   real(kind=c_double), dimension(n_jp) :: mj,rij
18   M_trgt = mass_avg * N_neighbor
19   do i=1,n_ip
20     dens = 0.0d0
21     h_max_alw = ep_i(i)%smth ! maximum allowance
22     h = h_max_alw / SCF_smth
23     ! Note that we increase smth by a factor of scf_smth
24     ! before calling calc_density().
25     h_L = 0.0d0
26     h_U = h_max_alw
27     dh_prev = 0.0d0
28     n_unchanged = 0
29     ! Software cache
30     do j=1,n_jp
31       mj(j) = ep_j(j)%mass
32       dr%x = ep_i(i)%pos%x - ep_j(j)%pos%x
33       dr%y = ep_i(i)%pos%y - ep_j(j)%pos%y
34       dr%z = ep_i(i)%pos%z - ep_j(j)%pos%z
35       rij(j) = dsqrt(dr%x * dr%x &
36                      + dr%y * dr%y &
37                      + dr%z * dr%z)
38     end do
39     iteration_loop: do
40       ! Calculate density
41       dens = 0.0d0
42       do j=1,n_jp
43         dens = dens + mj(j) * W(rij(j), h)
44       end do

```

```

45      ! Check if the current value of the smoothing length
46          satisfies
47      ! Eq.(5) in Springel (2005).
48      M = 4.0d0 * pi * h * h * h * dens / 3.0d0
49      if ((h < h_max_alw) .and. (dabs(M/M_trgt - 1.0d0) < eps))
50          then
51              ! In this case, Eq.(5) holds within a specified accuracy
52
53              f(i)%flag = 1
54              f(i)%dens = dens
55              f(i)%smth = h
56              exit iteration_loop
57      end if
58      if (((h == h_max_alw) .and. (M < M_trgt)) .or. (n_unchanged
59          == 4)) then
60          ! In this case, we skip this particle forcibly.
61          ! In order to determine consistently the density
62          ! and the smoothing length for this particle,
63          ! we must re-perform calcForceAllAndWriteBack().
64          f(i)%flag = 0
65          f(i)%dens = dens
66          f(i)%smth = h_max_alw
67          exit iteration_loop
68      end if
69      ! Update h_L & h_U
70      if (M < M_trgt) then
71          if (h_L < h) h_L = h
72      else if (M_trgt < M) then
73          if (h < h_U) h_U = h
74      end if
75      dh = h_U - h_L
76      if (dh == dh_prev) then
77          n_unchanged = n_unchanged + 1
78      else
79          dh_prev = dh
80          n_unchanged = 0
81      end if
82      ! Update smoothing length
83      h = ((3.0d0 * M_trgt)/(4.0d0 * pi * dens))**((1.0d0/3.0d0)
84      if ((h <= h_L) .or. (h == h_U)) then
85          ! In this case, we switch to the bisection search.
86          ! The inclusion of '=' in the if statement is very
87          ! important to escape a limit cycle.
88          h = 0.5d0 * (h_L + h_U)
89      else if (h_U < h) then
90          h = h_U
91      end if
92      end do iteration_loop
93      ! Calculate grad-h term
94      if (f(i)%flag == 1) then
95          drho_dh = 0.0d0
96          do j=1,n_jp
97              drho_dh = drho_dh + mj(j) * dWdh(rij(j), h)
98          end do
99          f(i)%gradh = 1.0d0 / (1.0d0 + (h * drho_dh) / (3.0d0 * dens))

```

```

)
96
97     f(i)%gradh = 1.0d0 ! dummy value
98 end if
99 ! Compute \div v & \rot v for Balsara switch
100 #if defined(USE_BALSARA_SWITCH)
101 do j=1,n_jp
102     dr%x = ep_i(i)%pos%x - ep_j(j)%pos%x
103     dr%y = ep_i(i)%pos%y - ep_j(j)%pos%y
104     dr%z = ep_i(i)%pos%z - ep_j(j)%pos%z
105     dv%x = ep_i(i)%vel%x - ep_j(j)%vel%x
106     dv%y = ep_i(i)%vel%y - ep_j(j)%vel%y
107     dv%z = ep_i(i)%vel%z - ep_j(j)%vel%z
108     gradW_i = gradW(dr, f(i)%smth)
109     f(i)%divv = f(i)%divv - mj(j) * (dv%x * gradW_i%x +
110                                         +dv%y * gradW_i%y +
111                                         +dv%z * gradW_i%z)
112     f(i)%rotv%x = f(i)%rotv%x - mj(j) * (dv%y * gradW_i%z - dv%z
113                                         * gradW_i%y)
114     f(i)%rotv%y = f(i)%rotv%y - mj(j) * (dv%z * gradW_i%x - dv%x
115                                         * gradW_i%z)
116     f(i)%rotv%z = f(i)%rotv%z - mj(j) * (dv%x * gradW_i%y - dv%y
117                                         * gradW_i%x)
118 end do
119 f(i)%divv = f(i)%divv / f(i)%dens
120 f(i)%rotv%x = f(i)%rotv%x / f(i)%dens
121 f(i)%rotv%y = f(i)%rotv%y / f(i)%dens
122 f(i)%rotv%z = f(i)%rotv%z / f(i)%dens
123 #endif
124 end do
125 #else
126     double precision :: mj,rij
127     do i=1,n_ip
128         f(i)%dens = 0.0d0
129         do j=1,n_jp
130             dr%x = ep_j(j)%pos%x - ep_i(i)%pos%x
131             dr%y = ep_j(j)%pos%y - ep_i(i)%pos%y
132             dr%z = ep_j(j)%pos%z - ep_i(i)%pos%z
133             rij = dsqrt(dr%x * dr%x +
134                           +dr%y * dr%y +
135                           +dr%z * dr%z)
136             f(i)%dens = f(i)%dens +
137                         + ep_j(j)%mass * W(rij,ep_i(i)%smth)
138         end do
139         f(i)%smth = ep_i(i)%smth
140         f(i)%gradh = 1.0d0
141         ! Compute \div v & \rot v for Balsara switch
142         #if defined(USE_BALSARA_SWITCH)
143             do j=1,n_jp
144                 mj = ep_j(j)%mass
145                 dr%x = ep_i(i)%pos%x - ep_j(j)%pos%x
146                 dr%y = ep_i(i)%pos%y - ep_j(j)%pos%y
147                 dr%z = ep_i(i)%pos%z - ep_j(j)%pos%z
148                 dv%x = ep_i(i)%vel%x - ep_j(j)%vel%x
149                 dv%y = ep_i(i)%vel%y - ep_j(j)%vel%y

```

---

```

147      dv%z = ep_i(i)%vel%z - ep_j(j)%vel%z
148      gradW_i = gradW(dr, f(i)%smth)
149      f(i)%divv = f(i)%divv - mj * (dv%x * gradW_i%x &
150                                +dv%y * gradW_i%y &
151                                +dv%z * gradW_i%z)
152      f(i)%rotv%x = f(i)%rotv%x - mj * (dv%y * gradW_i%z - dv%z *
153                                gradW_i%y)
154      f(i)%rotv%y = f(i)%rotv%y - mj * (dv%z * gradW_i%x - dv%x *
155                                gradW_i%z)
156      f(i)%rotv%z = f(i)%rotv%z - mj * (dv%x * gradW_i%y - dv%y *
157                                gradW_i%x)
158      end do
159      f(i)%divv = f(i)%divv / f(i)%dens
160      f(i)%rotv%x = f(i)%rotv%x / f(i)%dens
161      f(i)%rotv%y = f(i)%rotv%y / f(i)%dens
162      f(i)%rotv%z = f(i)%rotv%z / f(i)%dens
163 #endif
164     end do
165 #endif

```

---

#### 7.1.4.3 Interaction function for the calculation of pressure-gradient acceleration

Interaction function for the calculation of pressure-gradient acceleration is implemented as subroutine `calc_hydro_force`. Listing 62 shows its implementation. This performs the calculations of the right hand sides of Eqs.(10), (12), and (13), and `dt` according to Eq.(16) in Springel [2005, MNRAS, 364, 1105] (for `dt`, see the definition of `fp_sph` type ).

Listing 62: Interaction function for the calculation of pressure-gradient acceleration

---

```

1 !**** Interaction function
2 subroutine calc_hydro_force(ep_i,n_ip,ep_j,n_jp,f) bind(c)
3   integer(kind=c_int), intent(in), value :: n_ip,n_jp
4   type(ep_hydro), dimension(n_ip), intent(in) :: ep_i
5   type(ep_hydro), dimension(n_jp), intent(in) :: ep_j
6   type(force_hydro), dimension(n_ip), intent(inout) :: f
7   !* Local variables
8   integer(kind=c_int) :: i,j
9   real(kind=c_double) :: mass_i,mass_j,smth_i,smth_j, &
10                 dens_i,dens_j,pres_i,pres_j, &
11                 gradh_i,gradh_j,balsw_i,balsw_j, &
12                 snds_i,snds_j
13   real(kind=c_double) :: povrho2_i,povrho2_j, &
14                 v_sig_max,dr_dv,w_ij,v_sig,AV
15   type(fdps_f64vec) :: pos_i,pos_j,vel_i,vel_j, &
16                 dr,dv,gradW_i,gradW_j,gradW_ij
17   do i=1,n_ip
18     !* Zero-clear
19     v_sig_max = 0.0d0
20     !* Extract i-particle info.
21     pos_i = ep_i(i)%pos
22     vel_i = ep_i(i)%vel
23     mass_i = ep_i(i)%mass
24     smth_i = ep_i(i)%smth
25     dens_i = ep_i(i)%dens

```

---

```

26     pres_i = ep_i(i)%pres
27     gradh_i = ep_i(i)%gradh
28     balsw_i = ep_i(i)%balsw
29     snds_i = ep_i(i)%snds
30     povrho2_i = pres_i/(dens_i*dens_i)
31     do j=1,n_jp
32       !* Extract j-particle info.
33       pos_j%x = ep_j(j)%pos%x
34       pos_j%y = ep_j(j)%pos%y
35       pos_j%z = ep_j(j)%pos%z
36       vel_j%x = ep_j(j)%vel%x
37       vel_j%y = ep_j(j)%vel%y
38       vel_j%z = ep_j(j)%vel%z
39       mass_j = ep_j(j)%mass
40       smth_j = ep_j(j)%smth
41       dens_j = ep_j(j)%dens
42       pres_j = ep_j(j)%pres
43       gradh_j = ep_j(j)%gradh
44       balsw_j = ep_j(j)%balsw
45       snds_j = ep_j(j)%snds
46       povrho2_j = pres_j/(dens_j*dens_j)
47       !* Compute dr & dv
48       dr%x = pos_i%x - pos_j%x
49       dr%y = pos_i%y - pos_j%y
50       dr%z = pos_i%z - pos_j%z
51       dv%x = vel_i%x - vel_j%x
52       dv%y = vel_i%y - vel_j%y
53       dv%z = vel_i%z - vel_j%z
54       !* Compute the signal velocity
55       dr_dv = dr%x * dv%x + dr%y * dv%y + dr%z * dv%z
56       if (dr_dv < 0.0d0) then
57         w_ij = dr_dv / sqrt(dr%x * dr%x + dr%y * dr%y + dr%z * dr%z
58           )
59       else
60         w_ij = 0.0d0
61       end if
62       v_sig = snds_i + snds_j - 3.0d0 * w_ij
63       v_sig_max = max(v_sig_max, v_sig)
64       !* Compute the artificial viscosity
65       AV = - 0.5d0*v_sig*w_ij / (0.5d0*(dens_i+dens_j)) * 0.5d0*(
66         balsw_i+balsw_j)
67       !* Compute the average of the gradients of kernel
68       gradW_i = gradW(dr,smth_i)
69       gradW_j = gradW(dr,smth_j)
70       gradW_ij%x = 0.5d0 * (gradW_i%x + gradW_j%x)
71       gradW_ij%y = 0.5d0 * (gradW_i%y + gradW_j%y)
72       gradW_ij%z = 0.5d0 * (gradW_i%z + gradW_j%z)
73       !* Compute the acceleration and the heating rate
74       f(i)%acc%x = f(i)%acc%x - mass_j*(gradh_i * povrho2_i *
75         gradW_i%x +
76           +gradh_j * povrho2_j *
77             gradW_j%x &
78               +AV * gradW_ij%x)
79       f(i)%acc%y = f(i)%acc%y - mass_j*(gradh_i * povrho2_i *
80         gradW_i%y +
81           +gradh_j * povrho2_j *
82             gradW_j%y &
83               +AV * gradW_ij%y)
84

```

```

76                               +gradh_j * povrho2_j *
77                               gradW_j%y &
78 f(i)%acc%z = f(i)%acc%z - mass_j*(gradh_i * povrho2_i *
79                               gradW_i%z &
80                               +gradh_j * povrho2_j *
81                               gradW_j%z &
82                               +AV * gradW_ij%z)
83 f(i)%eng_dot = f(i)%eng_dot
84                               &
85                               + mass_j * gradh_i * povrho2_i * (dv%x * gradW_i%
86                               x &
87                               +dv%y * gradW_i%y &
88                               +dv%z * gradW_i%z) &
89 f(i)%ent_dot = f(i)%ent_dot
90                               + 0.5 * mass_j * AV * (dv%x * gradW_ij%x &
91                               +dv%y * gradW_ij%y &
92                               +dv%z * gradW_ij%z)
93 end do
94 f(i)%ent_dot = f(i)%ent_dot &
95 * (specific_heat_ratio - 1.0d0) &
96 / dens_i*(specific_heat_ratio - 1.0d0)
97 f(i)%dt = CFL_hydro*2.0d0*smth_i/v_sig_max
end do

```

---

### 7.1.5 Main body of the sample code

In this section, we describe the main body of the sample code implemented mainly in `f_main.F90`. Before entering a detailed explanation, we describe here the overall structure of the code. As described in the beginning of § 7.1, this code performs a  $N$ -body/SPH simulation of a disk galaxy. Thus, in the default, the code sets an initial condition for a disk galaxy. But, initial conditions for simple test calculations are also prepared in the code. More specifically, the code supports the following four types of initial conditions:

- (a) Initial condition for a disk galaxy simulation. It is selected when `-DINITIAL_CONDITION=0` is specified at the compile-time. The initial condition is created in subroutine `galaxy_IC` in `ic.F90`. The initial distributions of dark matter and star particles are set by reading a file created by MAGI. The initial distribution of gas (SPH) particles is determined in the subroutine. In the default, an exponential disk ( $M = 10^{10} M_{\odot}$ ,  $R_s = 7$  kpc [scale radius],  $R_t = 12.5$  kpc [truncation radius],  $z_d = 0.4$  kpc [scale height],  $z_t = 1$  kpc [truncation height]) is created with the number of SPH particles of  $2^{18}$ .
- (b) Initial condition for cold collapse test. It is selected when `-DINITIAL_CONDITION=1` is specified at the compile-time. The initial condition is created in subroutine `cold-collapse_test_IC` in `ic.F90`.

- (c) Initial condition for the Evrard test (§ 3.3 in [Evrard \[1988,MNRAS,235,911\]](#)). It is selected when `-DINITIAL_CONDITION=2` is specified at the compile-time. This initial condition is created in subroutine `Evrard_test_IC` in `ic.F90`. There are two options for the way of creating an initial condition. We can specify the way by manually set the value of the last argument of the function 0 or 1. If 0 is given, the function creates the density profile of the Evrard gas sphere by rescaling the positions of particles which are placed in a grid. If 1 is specified, it creates the density profile by rescaling the positions of particles which are distributed glass-like. In order to use the second option, we have to create particle data by executing the code with the mode described in the next item.
- (d) Operation mode to create a glass-like distribution of SPH particles in a box of  $[-1, 1]^3$ . This mode is selected when `-DINITIAL_CONDITION=3` is specified at the compile-time. The initial condition is created in subroutine `make_glass_IC` in `ic.F90`.

The structure of the sample code is as follows:

- (1) Create and initialize FDPS objects
- (2) Initialize the Phantom-GRAPE library for x86 if needed
- (3) Read a data file of  $N$ -body particles and make an initial condition
- (4) Calculate the motions of particles until the end time we specify

Below, we explain each item in detail.

#### 7.1.5.1 Creation of an object of type `fdps_controller`

In order to use APIs of FDPS, a user program should create an object of type `FDPS_controller`. In this sample code, `fdps_ctrl`, an object of type `FDPS_controller`, is created in the main routine.

Listing 63: Creation of an object of type `fdps_controller`

---

```

1 subroutine f_main()
2   use fdps_module
3   implicit none
4   !* Local variables
5   type(fdps_controller) :: fdps_ctrl
6
7   ! Do something
8
9 end subroutine f_main

```

---

Note that this code snippet only shows the necessary part of the code from the actual sample code. Also note that all FDPS APIs are called as member functions of this object because of the reason described above.

#### 7.1.5.2 Initialization and termination of FDPS

We need first to initialize FDPS by calling API `ps_initialize`:

Listing 64: Initialize FDPS

---

```

1 call fdps_ctrl%ps_initialize();

```

---

Once started, FDPS should be explicitly terminated by calling API `ps_finalize`. This sample code terminates FDPS just before the termination of the program. You can find the following code at the last part of `f_main.F90`.

Listing 65: Finalize FDPS

---

```
1 call fdps_ctrl%ps_finalize();
```

---

### 7.1.5.3 Creation and initialization of FDPS objects

After the initialization of FDPS, a user need to create the objects used to talk to FDPS. In this section, we describe how to create and initialize these objects.

#### 7.1.5.3.1 Creation and initialization of `ParticleSystem` objects

This sample code uses different `ParticleSystem` objects to manage  $N$ -body and SPH particles. Two integer variables `psys_num_nbody` and `psys_num_sph` are used to store the identification numbers for `ParticleSystem` objects for  $N$ -body and SPH particles, respectively. Using these variables, the creation and the initialization of the objects are done as follows.

Listing 66: Creation and initialization of `ParticleSystem` objects

---

```
1 call fdps_ctrl%create_psys(psys_num_nbody, 'fp_nbody')
2 call fdps_ctrl%init_psys(psys_num_nbody)
3 call fdps_ctrl%create_psys(psys_num_sph, 'fp_sph')
4 call fdps_ctrl%init_psys(psys_num_sph)
```

---

#### 7.1.5.3.2 Creation and initialization of `DomainInfo` object

This sample code decomposes the computational domain so that the *total* ( $N$ -body + SPH) particle distribution is divided equally. In this case, we need one `DomainInfo` object. Thus, using one integer variable `dinfo_num`, the creation and initialization of `DomainInfo` object are performed as follows.

Listing 67: Creation and initialization of `DomainInfo` object

---

```
1 call fdps_ctrl%create_dinfo(dinfo_num)
2 call fdps_ctrl%init_dinfo(dinfo_num, coef_ema)
```

---

#### 7.1.5.3.3 Creation and initialization of `TreeForForce` objects

The code uses three types of `TreeForForce` objects and they are used for the gravity calculation, the density calculation, and the calculation of pressure-gradient acceleration. When initializing a `TreeForForce` object, we must pass a typical number of particles used in the interaction calculation as the second argument of API `init_tree`. For `TreeForForce` object `tree_num_grav`, the value that is three times of the number of local particles ( $N$ -body + SPH) is passed. On the other hand, for `TreeForForce` objects `tree_num_dens` and `tree_num_hydro`, the value that is three times of the number of local SPH particles is passed.

Listing 68: Creation and initialization of TreeForForce objects

---

```

1 nptcl_loc_sph    = max(fdps_ctrl%get_nptcl_loc(psys_num_sph),1)
2 nptcl_loc_nbody = fdps_ctrl%get_nptcl_loc(psys_num_nbody)
3 nptcl_loc_all   = nptcl_loc_nbody + nptcl_loc_sph
4 !** tree for gravity calculation
5 call fdps_ctrl%create_tree(tree_num_grav, &
6                             "Long,force_grav,ep_grav,ep_grav,Monopole")
7 call fdps_ctrl%init_tree(tree_num_grav, 3*nptcl_loc_all, theta, &
8                          n_leaf_limit, n_group_limit)
9 !** tree for the density calculation
10 call fdps_ctrl%create_tree(tree_num_dens, &
11                            "Short,force_dens,ep_hydro,ep_hydro,Gather")
12 call fdps_ctrl%init_tree(tree_num_dens, 3*nptcl_loc_sph, theta, &
13                          n_leaf_limit, n_group_limit)
14 !** tree for the hydrodynamic force calculation
15 call fdps_ctrl%create_tree(tree_num_hydro, &
16                            "Short,force_hydro,ep_hydro,ep_hydro,
17                                         Symmetry")
18 call fdps_ctrl%init_tree(tree_num_hydro, 3*nptcl_loc_sph, theta, &
19                          n_leaf_limit, n_group_limit)

```

---

#### 7.1.5.4 Setting initial condition

The initial condition is set in subroutine `setup_IC`, which internally calls a different subroutine depending on the value of macro `INITIAL_CONDITION`. The correspondence relation between the name of a internally-called subroutine and the value of the macro has been described already in the beginning part of § 7.1.5. The arguments `time_dump`, `dt_dump`, `time_end` represents the initial time of data output, the time interval of data output, and the end time of the simulation, respectively. These must be set in this subroutine. Also, the boundary condition, the gravitational softening (`eps_grav`), the maximum allowable time step of the system (`dt_max`) are set in this subroutine (a user does not necessarily set `dt_max`).

Listing 69: Setting initial condition

---

```

1 call setup_IC(psys_num_nbody, psys_num_sph, dinfo_num, &
2               time_dump, dt_dump, time_end);

```

---

In what follows, we describe some of points to remember for subroutine `galaxy_IC`.

- MAGI outputs particle data in its code unit. The information about the MAGI’s code unit is described in file `./magi_data/doc/unit.txt` (see section “Computational unit”). This file is created when executing MAGI. The variables `magi_unit_mass`, `magi_unit_leng`, `magi_unit_time` in the subroutine must be consistent with the MAGI’s code unit.
- The subroutine reads particle data from file of the name of `./magi_data/dat/Galaxy.tipsy` in the default. If you make the code read a different file, please change the source code manually.

- The subroutine generates an initial gas distribution which has exponential profile along both  $R$  ( $\equiv \sqrt{x^2 + y^2}$ ) and  $z$  directions. The variables `Rs` and `zd` represents the scale lengths. The variables `Rt` and `zt` represents the truncation (cutoff) lengths.
- The initial thermodynamic state is specified by both the initial gas temperature `temp` and the mean molecular weight relative to the mass of hydrogen atom `mu`. Regardless of the state of the macro `USE_ENTROPY`, a user must specify the thermodynamic state of SPH particles via the specific internal energy (member variable `eng` in `fp_sph` type )[the sample code automatically does this]. If the macro `USE_ENTROPY` is defined, the initial value of the entropy is automatically set by subroutine `set_entropy` called in the subroutine `f_main()`, using the initial value of the specific internal energy and the calculated density. On the other hand, if the macro is not defined, the value of `eng` set in the subroutine `galaxy_IC` is treated as the initial value of the specific internal energy.

#### 7.1.5.5 Domain decomposition

When there are different types of `ParticleSystem` objects, the domain decomposition based on the combined distribution of particles can be realized by using APIs `collect_sample_particle` and `decompose_domain`. First, a user have to collect sample particles from each `ParticleSystem` object using API `collect_sample_particle`. Here, we must pass `.false.` to the third argument of this API for the second or later `ParticleSystem` object because the previous information is cleared without this. After collecting sample particles from all of `ParticleSystem` objects, call API `decompose_domain` to perform domain decomposition.

Listing 70: Domain decomposition

---

```
1 call fdps_ctrl%collect_sample_particle(dinfo_num, psys_num_nbody, clear)
2 call fdps_ctrl%collect_sample_particle(dinfo_num, psys_num_sph, unclear)
3 call fdps_ctrl%decompose_domain(dinfo_num)
```

---

#### 7.1.5.6 Particle exchange

In order to perform particle exchange based on the previous-calculated domain information, it is only necessary to call API `exchange_particle`.

Listing 71: Particle exchange

---

```
1 call fdps_ctrl%exchange_particle(psys_num_nbody, dinfo_num)
2 call fdps_ctrl%exchange_particle(psys_num_sph, dinfo_num)
```

---

#### 7.1.5.7 Interaction calculations

After the domain decomposition and particle exchange, interaction calculations are done. Below, we show the implementation of the interaction calculations just after setting the initial condition. At first, the code performs the gravity calculation. Then, it performs the calculations of density and pressure-gradient acceleration.

Listing 72: Interaction calculations

---

```

1  !*** Gravity calculation
2  t_start = fdps_ctrl%get_wtime()
3 #if defined(ENABLE_GRAVITY_INTERACT)
4  call fdps_ctrl%set_particle_local_tree(tree_num_grav, psys_num_nbody)
5  call fdps_ctrl%set_particle_local_tree(tree_num_grav, psys_num_sph,
6      unclear)
7  pfunc_ep_ep = c_funloc(calc_gravity_ep_ep)
8  pfunc_ep_sp = c_funloc(calc_gravity_ep_sp)
9  call fdps_ctrl%calc_force_making_tree(tree_num_grav, &
10      pfunc_ep_ep, &
11      pfunc_ep_sp, &
12      dinfo_num)
13 nptcl_loc_nbody = fdps_ctrl%get_nptcl_loc(psys_num_nbody)
14 call fdps_ctrl%get_psyst_fptr(psys_num_nbody, ptcl_nbody)
15 do i=1,nptcl_loc_nbody
16     call fdps_ctrl%get_force(tree_num_grav, i, f_grav)
17     ptcl_nbody(i)%acc%x = f_grav%acc%x
18     ptcl_nbody(i)%acc%y = f_grav%acc%y
19     ptcl_nbody(i)%acc%z = f_grav%acc%z
20     ptcl_nbody(i)%pot    = f_grav%pot
21 end do
22 offset = nptcl_loc_nbody
23 nptcl_loc_sph = fdps_ctrl%get_nptcl_loc(psys_num_sph)
24 call fdps_ctrl%get_psyst_fptr(psys_num_sph, ptcl_sph)
25 do i=1,nptcl_loc_sph
26     call fdps_ctrl%get_force(tree_num_grav, i + offset, f_grav)
27     ptcl_sph(i)%acc_grav%x = f_grav%acc%x
28     ptcl_sph(i)%acc_grav%y = f_grav%acc%y
29     ptcl_sph(i)%acc_grav%z = f_grav%acc%z
30     ptcl_sph(i)%pot_grav  = f_grav%pot
31 end do
32 #endif
33 t_grav = fdps_ctrl%get_wtime() - t_start
34 !*** SPH calculations
35 t_start = fdps_ctrl%get_wtime()
36 #if defined(ENABLE_HYDRO_INTERACT)
37 call calc_density_wrapper(psys_num_sph, dinfo_num, tree_num_dens)
38 call set_entropy(psys_num_sph)
39 call set_pressure(psys_num_sph)
40 pfunc_ep_ep = c_funloc(calc_hydro_force)
41 call fdps_ctrl%calc_force_all_and_write_back(tree_num_hydro, &
42     pfunc_ep_ep, &
43     psys_num_sph, &
44     dinfo_num)
45 #endif
46 t_hydro = fdps_ctrl%get_wtime() - t_start

```

---

First, we explain the part of the implementation for the gravity calculation. In the gravity calculation, both  $N$ -body and SPH particles are involved. In order to perform an interaction calculation between different types of particles, we must use in combination TreeForForce object's APIs `set_particle_local_tree` and `calc_force_making_tree`. We first pass the particle information stored in each ParticleSystem object to a TreeForForce object using API `set_particle_local_tree`. Here, we must pass `.false.` to the third

argument of this API for the second or later ParticleSystem objects because all of the previously-passed information is cleared without this. After finishing calling this API for all of ParticleSystem objects that are involved in the gravity calculation, call API `calc_force_making_tree` to perform the interaction calculation. In order to obtain the result of the interaction calculation, we need to use API `get_force`. This API takes an integral argument  $i$ , and it writes the force of the  $i$ th particle read by API `set_particle_local_tree` in the address specified by the third argument of the API. Hence, we must use appropriate offset to obtain the results of the interaction calculation of the second or later ParticleSystem.

Next, we explain the part of the implementation for the calculations of density and pressure-gradient acceleration. These interaction calculations involves only single type of particles, SPH particles. Therefore, we can use API `calc_force_all_and_write_back`, which is frequently used in the sample code introduced in this document. For the calculation of pressure-gradient acceleration, the code performs this API in the subroutine `f_main()`. On the other hand, we need to handle the case that the iteration calculation of  $\rho_i$  and  $h_i$  does not converge for some particles as described in § 7.1.4. This handling is done in the subroutine `calc_density_wrapper`. The implementation of this subroutine is shown below. The implementation actually used differs depending on the state of the macro `ENABLE_VARIABLE_SMOOTHING_LENGTH`. If it is not defined, the code calls API `calc_force_all_and_write_back` only once because in this case the code performs SPH calculation as the fixed smoothing length SPH code. If the macro is defined, the code calls the API repeatedly until  $\rho_i$  and  $h_i$  of all the particles are self-consistently determined. The member variable `flag` stores the result of the iteration calculation and the value of 1 means that the iteration converges successfully. So, the code stops the infinite `do-enddo` loop when the number of SPH particles whose `flag` has the value of 1 agrees with the total number of SPH particles.

Listing 73: Subroutine `calc_density_wrapper`

---

```

1 subroutine calc_density_wrapper(psyst_num, dinfo_num, tree_num)
2   use fdps_vector
3   use fdps_module
4   use user_defined_types
5   implicit none
6   integer, intent(in) :: psyst_num, dinfo_num, tree_num
7   !* Local variables
8   integer :: i, nptcl_loc, nptcl_glb
9   integer :: n_compl_loc, n_compl
10  type(fdps_controller) :: fdps_ctrl
11  type(fp_sph), dimension(:), pointer :: ptcl
12  type(c_funptr) :: pfunc_ep_ep
13
14 #if defined(ENABLE_VARIABLE_SMOOTHING_LENGTH)
15  nptcl_loc = fdps_ctrl%get_nptcl_loc(psyst_num)
16  nptcl_glb = fdps_ctrl%get_nptcl_glb(psyst_num)
17  call fdps_ctrl%get_psyst_fptr(psyst_num, ptcl)
18  pfunc_ep_ep = c_funloc(calc_density)
19  ! Determine the density and the smoothing length
20  ! so that Eq.(6) in Springel (2005) holds within a specified accuracy.
21  do
22    ! Increase smoothing length
23    do i=1, nptcl_loc

```

---

```

24         ptcl(i)%smth = scf_smth * ptcl(i)%smth
25     end do
26     ! Compute density, etc.
27     call fdps_ctrl%calc_force_all_and_write_back(tree_num,      &
28                                         pfunc_ep_ep,      &
29                                         psys_num,        &
30                                         dinfo_num)
31     ! Check convergence
32     n_compl_loc = 0; n_compl = 0
33     do i=1,nptcl_loc
34         if (ptcl(i)%flag == 1) n_compl_loc = n_compl_loc + 1
35     end do
36     call fdps_ctrl%get_sum(n_compl_loc, n_compl)
37     if (n_compl == nptcl_glb) exit
38   end do
39   !* Release the pointer
40   nullify(ptcl)
41 #else
42   pfunc_ep_ep = c_funloc(calc_density)
43   call fdps_ctrl%calc_force_all_and_write_back(tree_num,      &
44                                         pfunc_ep_ep,      &
45                                         psys_num,        &
46                                         dinfo_num)
47 #endif
48
49 end subroutine calc_density_wrapper

```

---

subroutine `set_entropy` is called only once just after setting an initial condition. As described earlier, this subroutine is used to set the initial value of the entropy. Because we need the initial density to set the initial value of the entropy using Eq. (8), this subroutine is placed just after subroutine `calc_density_wrapper`. After this, the entropy becomes the independent variable to describe the thermodynamic state of gas if the macro `USE_ENTROPY` is defined.

#### 7.1.5.8 Time integration

This code performs the time integration using the Leapfrog method (see § 4.1.3.5.4 for this method). In this code,  $D(\cdot)$  operator is implemented as the subroutine `full_drift`, while  $K(\cdot)$  operator is implemented as subroutines `initial_kick` and `final_kick`.

## 8 User Supports

We accept questions and comments on FDPS at the following mail address:  
[fdps-support@mail.jmlab.jp](mailto:fdps-support@mail.jmlab.jp)

Please provide us with the following information.

### 8.1 Compile-time problem

- Compiler environment (version of the compiler, compile options etc)
- Error message at the compile time
- (if possible) the source code

### 8.2 Run-time problem

- Run-time environment
- Run-time error message
- (if possible) the source code

### 8.3 Other cases

For other problems, please do not hesitate to contact us. We sincerely hope that you'll find FDPS useful for your research.

## 9 License

This software is MIT licensed. Please cite Iwasawa et al. (2016, Publications of the Astronomical Society of Japan, 68, 54) and Namekata et al. (2018, Publications of the Astronomical Society of Japan, 70, 70) if you use the standard functions only.

The extended feature “Particle Mesh” is implemented by using a module of GREEM code (Developers: Tomoaki Ishiyama and Keigo Nitadori) (Ishiyama, Fukushige & Makino 2009, Publications of the Astronomical Society of Japan, 61, 1319; Ishiyama, Nitadori & Makino, 2012 SC’12 Proceedings of the International Conference on High Performance Computing, Networking Storage and Analysis, No. 5). GREEM code is developed based on the code in Yoshikawa & Fukushige (2005, Publications of the Astronomical Society of Japan, 57, 849). Please cite these three literatures if you use the extended feature “Particle Mesh”.

Please cite Tanikawa et al.(2012, New Astronomy, 17, 82) and Tanikawa et al.(2012, New Astronomy, 19, 74) if you use the extended feature “Phantom-GRAPE for x86”.

Copyright (c) <2015-> <FDPS developer team>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.